Point Based Graphics Rendering with Unified Scalability Solutions

Lee Bull B.Sc. M.Sc.

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy
of the
University of London.

Department of Computer Science
University College London

August, 2006

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

Standard real-time 3D graphics rendering algorithms use brute force polygon rendering, with complexity linear in the number of polygons and little regard for limiting processing to data that contributes to the image. Modern hardware can now render smaller scenes to pixel levels of detail, relaxing surface connectivity requirements.

Sub-linear scalability optimizations are typically self-contained, requiring specific data structures, without shared functions and data.

A new point based rendering algorithm 'Canopy' is investigated that combines multiple typically sub-linear scalability solutions, using a small core of data structures. Specifically, *locale management*, *hierarchical view volume culling*, *backface culling*, *occlusion culling*, *level of detail* and *depth ordering* are addressed. To demonstrate versatility further, shadows and collision detection are examined.

Polygon models are voxelized with interpolated attributes to provide points. A *scene tree* is constructed, based on a BSP tree of points, with compressed attributes. The scene tree is embedded in a compressed, partitioned, procedurally based scene graph architecture that mimics conventional systems with groups, instancing, inlines and basic *read on demand* rendering from backing store.

Hierarchical scene tree refinement constructs an *image tree* image space equivalent, with object space *scene node* points projected, forming *image node* equivalents. An *image graph* of image nodes is maintained, describing image and object space occlusion relationships, hierarchically refined with front to back ordering to a specified threshold whilst occlusion culling with occluder fusion. Visible nodes at medium levels of detail are refined further to rasterization scales. Occlusion culling defines a set of visible nodes that can support caching for temporal coherence.

Occlusion culling is approximate, possibly not suiting critical applications. Qualities and performance are tested against standard rendering.

Although the algorithm has a $O(f^2)$ upper bound in the scene size f, it is shown to practically scale sub-linearly. Scenes with several hundred billion polygons conventionally, are rendered at interactive frame rates with minimal graphics hardware support.

Dedication

To my Grandma and Grandad who raised me.

Remembered always, with great love and thanks.

Acknowledgements

Thanks to my supervisor Mel Slater for his guidance, patience and support.

Acknowledgements also go to the EPSRC who funded the first years of this research.

Thanks also to my wife Jarka and family.

Contents

List of Figures

List of Tables

Chapter 1 *Introduction*

This thesis describes a real-time 3D rendering algorithm with typical sub-linear scalability that combines established forms of scalability solutions into a tightly unified architecture that includes locale management, view volume culling, back face culling, occlusion culling, shadows and collision detection. This means that the system's running time scales better than linearly with increasing scene size and detail. The algorithm is approximate and based on splatting, without the use of polygons. The system is shown to scale far better than linearly in tests and renders scenes that would conventionally consist of several hundred billion polygons at interactive frame rates.

This chapter ***introduces the thesis*** in Section 1.1, with a summary of ***standard 3D rendering*** in Section 1.2 and Section 1.3.

Scalability issues addressed in this thesis are examined in Section 1.4, with a proposed framework for the algorithm presented in this thesis.

The ***scope*** is defined for the thesis in Section 1.5. The ***contributions*** considered to be made are given in Section 1.6, with an ***overview of the rendering solution*** in Section 1.7.

Finally, the ***structure of this thesis*** is outlined in Section 1.8.

1.1 Rendering 3D Scenes

This thesis investigates a real-time rendering architecture that typically scales better than linearly with increasing scene size and detail, a property termed *sub-linear* scalability. It is designed specifically for very highly geometrically detailed environments of very large sizes. Moreover, it aims to unify multiple sub-linear scalability solutions through shared data structures and sub-tasks. An approach characterised as Point Based Rendering (PBR) is used. Unlike most PBR research, an emphasis is placed on the system as a general architecture for large scenes, rather than on quality rendering of single objects. The approach is limited to local illumination models, but could be extended to others such as ray tracing [70] [212] to add specular, mirrored reflections. Static scenes are addressed, but the architecture does not prohibit further development for dynamics.

The geometrical and graphical representation of the 3D world has a wide range of uses, due to its ability to mimic the environment in which we live. Moreover, it is a useful testing ground for environments that we cannot, or do not wish to create, due to limitations of physical ability, economy, time or safety. Today, we see notable applications in film and television production, medical imaging, Computer Aided Design (CAD) and Manufacture (CAM), visualization, architecture, games, simulation and general communication of information.

Rendering is the process of mathematically transforming a numerical representation of a 3D scene, to form a 2D image that represents it. The field of graphics has evolved from the earliest interactive rendering in Sutherland's Sketchpad application [197] to 3D scenes of far more richness, quality and versatility, that when intended, are capable of photo realism that is indistinguishable from reality to most viewers. Yet modern computing hardware that we know to be very fast indeed, is still several orders of magnitude from achieving rendering that we'd characterise as very high quality, at the speed at which we can perceive and interact with it, in 'real-time', often considered to be at least 20 frames per second [207]. Historically, this has led to a partition in the field, between off-line higher quality and real-time lower quality rendering.

Off-line rendering typically takes the time needed to produce the quality of result required, though this is still often subject to time constraints, for example in the film industry where production schedules are still notably affected by rendering times. Often,

off-line rendering techniques are able to consider light interaction with surfaces more accurately, resulting in more complex images that can, if necessary, simulate real lighting conditions using global illumination techniques such as radiosity [73] that model light transport between multiple surfaces before entering the virtual camera. Real-time applications have always been driven by their speed requirements, typically producing frames at 20Hz or faster, but the quality and versatility has been seen to increase, particularly driven by custom 3D rendering hardware developed initially for high end visualization and latterly for the games industry. Local illumination [63] models where light interaction between simple light sources and illuminated surfaces is the most typical approach in real-time systems and only a local approach will be considered in this thesis.

Over time, as generic processing hardware becomes faster, we inevitably see application types crossing this divide from the off-line realm to the real-time. There is still substantial pressure for performance improvement in both off-line and real-time applications. Both areas will benefit from increased scene complexity and rendering quality, whilst meeting their respective time constraints. Technological advances in one, may well be applicable to the other.

Light propagation in the scene, geometrical detail and scene size are the primary aspects that affect rendering performance. As this thesis uses a simple, fast local illumination model, the remaining factor affecting performance is the amount of scene geometry. The real world can be considered infinitely large and infinitely detailed for the purposes of scene representation and rendering and ideally, an algorithm would not be sensitive to the amount of data. The algorithm would also not prohibit the kind of dynamics possible in the real world, such as the ability for objects to move, be deformable and for lighting conditions to change.

Real-time rendering engines (see Section 1.3) have historically been based on relatively simple polygon based scene representation and rendering techniques, supported by a hardware 3D pipeline (see Section 1.2) to draw high numbers of graphical primitives with surface shading due to material properties and light sources. This approach is reliable, but grounded on a brute force approach with linear scalability in the scene size. As such, the software and hardware industry has tended to base its designs around this, isolated from any risk of more complex techniques, whilst advanced by successive incre-

ments in processing power offered by Central Processing Units (CPUs) and more recently, Graphics Processing Units [60] (GPUs).

However, much research has been undertaken to realise sub-linear scalability, usually based on pre-computation, identification and removal of redundant processing or through techniques that approximate a solution. This often comes at the cost of increased restrictions, implementation difficulty and perhaps, memory use, correctness and robustness of solution. Culling is an effective approach that rapidly identifies regions that are not visible to the viewer, such that they need not be process any further. Level of detail (LOD) control is also a very effective tool that reduces detail where it may be less required. Coherence, or similarity between views, scene states or intermediate calculations can also be used to increase performance, *image based rendering* being one such example. A large range of techniques is discussed later in Chapter 2.

Rendering algorithms generally have a particular set of properties that are of concern. These include *rendering speed, image quality, surface and lighting dynamic potential, memory use* and *scalability.* These respectively address performance, quality in terms of fitness for the application, the ability for objects, surfaces and lighting conditions to change state, the memory overheads required and how the algorithm scales with scene size. Rendering algorithms differ in that they trade off one or more of these characteristics, to gain one or more others. Optimization is primarily concerned with attempting to advance one or more of these properties, whilst sacrificing those that remain, in a way that is acceptable for the intended application. In terms of a given set of requirements for a rendering algorithm, there are no commonly agreed solutions that are considered to be optimal.

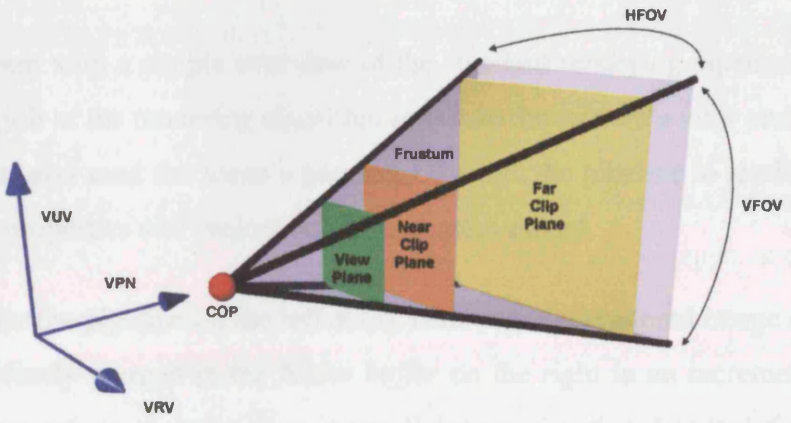
Although there has been substantial research in sub-linear rendering techniques, it has often concentrated on single aspects of standard linear rendering for improvement, in isolation of other parts of the system. As such, these optimizations only serve to improve the existing approach to real-time rendering, rather than re-considering the problem in its entirety, particularly in light of any new opportunities that may now be afforded by increased processing and memory resources. It is also now a common observation that polygon sizes in models are approaching pixel sizes, raising questions as to whether any restrictions and processing overheads resulting from their use, outweigh their benefits.

1.2 Standard Rendering Pipeline

This section overviews the standard rendering pipeline present in most real-time software and hardware implementations. How the pipeline is used is then discussed in Section 1.3. Polygon surface representations are most common, but other primitives such as points and lines are also sometimes used [63].

Polygonal scenes are generally represented using hierarchies of transformations with object surface descriptions at the leaves. This data structure is commonly referred to as a *scene graph*. VRML97 [203], Java3D [193], Open Inventor [209] and SGI Performer [188] are examples of standard systems structured in this way. Surface descriptions consist of indexed face sets with associated material information and references to texture maps with parameters for their application. Scene graphs typically contain other information pertaining to the virtual camera and light source positions.

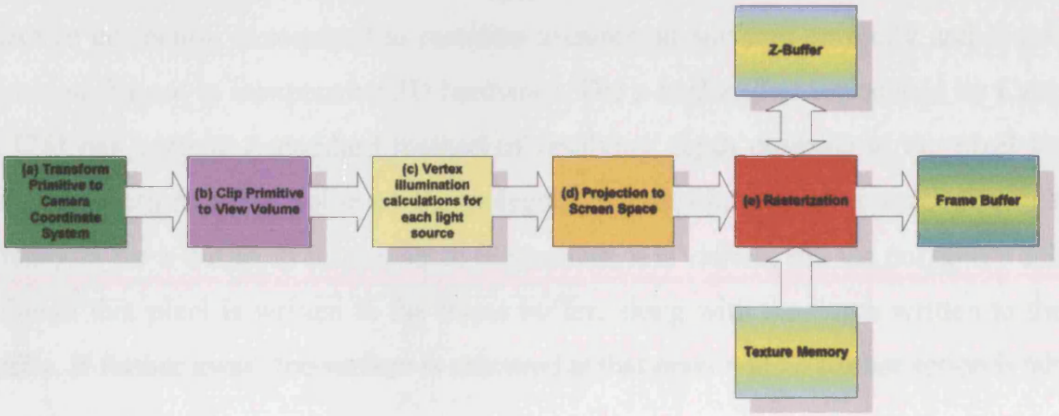
FIGURE 1. Virtual camera model with frustum used by many rendering methods



A typical pinhole camera model is shown in Figure 1, with *a frustum* view volume, a capped pyramidal volume that defines the field of view and depth range, within which, objects are considered for rendering. The camera is located in a parent coordinate system, with its view volume defined within its own orthogonal coordinate system, often referred to as *camera space*, shown by the three axes. The *view plane normal* (VPN) dictates the direction in which the camera points, i.e. the *pitch* and *yaw*. The *View Up Vector*

(VUV) is perpendicular to the VPN and dictates the camera's *roll*. The *centre of projection* (COP) is the point at which all rays which enter the camera converge in a perspective projection. This point is behind the *view plane* on which the rendered image is projected. *Near* and *far* clipping planes bound the view frustum which spans the horizontal and vertical fields of view. Alternatively shaped view volumes are possible, such as more complex abstract shapes or cones if required.

FIGURE 2. Classical polygonal rendering pipeline block diagram



A block diagram with a simple overview of the standard rendering pipeline is shown in Figure 2. The job of the rendering algorithm is to take the camera's view and the scene in its current state and send the scene's geometry through the pipeline to render the image, subject to the properties and projection of the camera model.

Primitives enter the pipeline on the left at (a), resulting in a rendered image of the primitive being correctly merged in the *frame buffer* on the right in an incremental fashion. Pipeline stages perform their function in parallel, assuming that data is defined for them to work on, by the previous stage. An initial setup stage is required to fill the pipeline to establish peak throughput, incurring a setup delay if the pipeline is ever flushed.

The first stage (a) transforms a 3D graphics primitive into the camera's coordinate system ready for projection onto the camera's view plane. Primitives are then clipped in (b) to the view volume in camera space. If they intersect the view volume planes, their geometry is clipped to lie on the volume. In the case of polygons, the clipping process may result in polygons being subdivided into smaller polygons. Illumination calculations may then performed for the vertices in camera space at (c) before undergoing the chosen

projection to the view plane in (d), which is generally a perspective or parallel projection. Once projected, the coordinates also undergo a conversion from device independent coordinates to device dependent coordinates that are dependent on the frame buffer's pixel dimensions. Stages (a) to (d) are sometimes referred to as the *setup engine* or *vertex processor* in hardware, as they set up the primitive ready for rasterization. Triangles are typically supported because convex polygons with more sides can be trivially broken down to triangles either by the rendering algorithm or the pipeline. Finally, the primitive is rasterized to the frame buffer with any defined textures and per-pixel shading. Perspective correction is required to rasterize textures on surfaces correctly and is now a common feature in inexpensive 3D hardware. The *z-buffer*, first introduced by Catmull in [25] has become a standard method of resolving depth ordering at the pixel level. After projection, the z value of the polygon is interpolated and compared with that already in the z-buffer. If it is closer to the camera, it is visible, and the polygon's colour value at that pixel is written to the frame buffer, along with the depth written to the z-buffer. If further away, the surface is obscured at that pixel and no further action is taken.

Hardware pipeline systems now incorporate programmable vertex and pixel shaders that execute compiled higher level languages such as nVidia's Cg [60], which correspond to the triangle setup phase and rasterization phase respectively. Shading and other operations can be performed per vertex or per pixel, incorporating a range of external parameters such as arbitrary values that may represent data such as light sources, and a constrained set of pipeline internal parameters.

Several basic shading methods are commonly used. Flat shading [63], samples shading for the whole primitive that is rasterized using the resulting colour. Gouraud [63] shading samples shading at vertices and interpolates across the primitive. Phong shading [63] samples illumination at each pixel and is now the most common form of shading in use. More complex functions can be implemented to achieve specific material appearances and special effects [60] [63].

Multiple rendering pipelines can be created within a single system because the process is separable until buffer compositing at the end of the pipeline. To date, as many as 48 pipelines are included in ATI's graphics processing unit (GPU) found in the Microsoft XBOX 360.

1.3 Standard Rendering Algorithms

This section discusses the standard type of rendering algorithm found in rendering systems such as VRML97 [203], Java3D [193], Open Inventor [209] and SGI Performer [188] that use the pipeline discussed in Section 1.2. Scalability issues of these algorithms and a framework for a solution are then discussed in Section 1.4.

Standard rendering algorithms use the rendering pipeline, typically to render polygon based models for real-time or interactive applications, where the time taken to generate each image is of greater concern than realism or whatever the ideal image qualities are for the application. Polygon rendering systems are efficient in a number of respects. Their representation is compact and explicit in that they are a linear interpolation of sample points on surfaces. Normally it will be an approximation of the surface, particularly if curved, though it is possible to represent shapes exactly, e.g. for faceted Computer Aided Design (CAD) projects intended for automated manufacture. More complex primitives such as curved Bezier, B-spline or NURBS surfaces [63] are sometimes reduced to polygon primitives before entering the pipeline or may be catered for explicitly within a more complex pipeline, but these forms will not be considered here.

The main rendering algorithm uses the 3D pipeline described in Section 1.2 to render primitives that represent the surfaces of objects in the scene. The most basic form of rendering algorithm would pass all of the scene's geometric primitives through the pipeline and let the pipeline discard scene detail that does not contribute to the image. This is, however, a very inefficient approach. Rendering algorithms in widespread use generally employ a method of selecting a subset of objects in the scene that may be visible in the image, as quickly as possible, that are then sent through the pipeline. Some of these simple methods will be discussed here.

A common technique is to compare an object's bounding volume, such as a sphere or box that contains the object's primitives, against the view volume [63]. If the bounding volume intersects with, or is contained by the view volume, the object is potentially visible and its primitives are sent through the pipeline. This test is typically much faster than passing the primitives through the pipeline, only to be view volume culled because they are outside the volume. For a scene S with n discrete objects, this view volume culling

method scales linearly with upper bound $O(n)$ and lower bound $\Omega(n)$. In complexity theory notation, $g(n) = O(f(n))$ indicates that $cf(n)$ is an upper bound on the running time of function $g(n)$, where c is a constant and similarly, $g(n) = \Omega(f(n))$ indicates that $cf(n)$ is a lower bound on the running time of $g(n)$.

To improve on linear view volume culling, hierarchical bounding volumes are often formed, based around the hierarchy resulting from the object's design. Bounding volumes are tested down the hierarchy, culling a volume and implicitly its child volumes if outside the view volume. A hierarchy of bounding volumes can achieve a $\Omega(1)$ lower bound, in the case where the root node is culled. However, the average complexity is unlikely to approach $O(\log n)$ because the structure of the hierarchy is dependent on the scene's design, which may be subject to content driven criteria rather than the formation of tight bounding volumes and balanced hierarchy. The upper bound $O(n)$ remains linear due to the case of a completely skew hierarchy.

Once objects are identified as potentially visible, their primitives are sent through the pipeline. A common speedup that can be implemented either in rendering algorithms or in the pipeline itself, is *back face culling* [63], whereby primitives such as polygons are discarded if their front side is not visible to the camera. This only works for polyhedral objects because their boundary representation surfaces do not need to be viewed from both sides.

It can be seen that these forms of standard rendering algorithm are linear or typically between linear and logarithmic in the number of discrete objects in the scene, for view volume culling. The sequential nature of the pipeline also means that it is linear in the number of primitives sent through.

A wide range of optimizations are discussed in Chapter 2. The next section discusses the substantial scalability issues of standard rendering, whilst deriving an algorithm framework for achieving practical sub-linear scalability.

1.4 Rendering Problems and Solutions

This section looks at the problems of standard rendering discussed in Section 1.2 and Section 1.3, whilst also deriving a framework for a solution, based on successive reductions in scene data. Scalability issues are addressed to explain why basic rendering algorithms do not scale to very large, detailed scenes. In particular, we examine what happens when a scene is theoretically infinite in both size and detail. For each problem, a solution type is identified to form a framework for the *Canopy* algorithm investigated.

Polygon based rendering is generally the de-facto standard used by real-time rendering algorithms and pipeline implementations. The piece-wise linear polygon surface representation has been historically required because until recent years, hardware and especially software rendering pipelines have been incapable of processing large numbers of polygons at real-time rates. As more polygons are used to describe an object, the polygons themselves typically become smaller. For many types of objects, particularly types with curves such as those that frequently occur in nature, small polygons are required to accurately sample the required surface.

A balance must be struck between the time taken to process vertices and time taken to rasterize. As more triangles are used in a surface, greater speed is required in the vertex processor. Actual total rasterization area in the image does not necessarily increase notably, but more time will also be used for rasterization due to additional overheads of rendering polygon edges. Objects at a distance from the viewpoint with a fixed level of detail may still have polygons that are sub-pixel in size when rasterized, making the system less efficient. Even though the primitive filling operation is expensive in itself, for polygons that are large in image space, it is still less costly than the more complex processing required to project one or more primitives to each pixel with per pixel shading. Until the late 1990s, hardware was not capable of performing this shift because vertex processors were still slow. Therefore, larger polygons were typically used, linearly interpolating projected vertices in image space.

Current hardware is now capable of rendering near pixel, pixel or even sub-pixel sized polygons under projection, to the frame buffer for small, typical scenes with viewpoints

close to surfaces, in real-time. This raises the question as to whether the polygon surface representation is still the best option.

A highly scalable and efficient local illumination rendering algorithm would only address scene objects and primitives that contribute to the image because they are directly visible. Basic rendering algorithms of the type described in Section 1.3 have several substantial performance, scalability and logistic issues, regardless of the pipeline, with respect to:

1. Addressing of objects for view volume visibility testing
2. Fixed surface detail
3. Rendering of all occluded surfaces
4. Pixel over-draw to resolve depth ordering
5. Representation inconsistency at differing scales

To begin, we will examine the first three of these scalability problems with respect to scene size and detail. The first aspect will consider reducing the set of objects to be addressed to just those that are potentially visible in the view volume. The second considers limitations on the amount of detail addressed. The third considers limiting the addressing of objects or surface regions in the view volume to only those that are visible and not fully occluded by those closer to the viewpoint. Pixel over-draw issues and representation inconsistency will then be discussed.

Consider a scene S consisting of n discrete objects. Standard rendering algorithms address objects to test them for potential visibility within the camera's view volume, typically using bounding volume tests described in Section 1.3. A linear $O(n) = \Omega(n)$, view volume visibility algorithm is formed if all n objects in the scene are subjected to a view volume culling test. A logarithmic lower bound $\Omega(\log n)$ can be introduced if hierarchical bounding volumes are used, improving the average complexity practically achieved, dependent on the efficient balancing of the tree describing the hierarchy.

In the theoretical case of $n = \infty$, where the scene is of infinite size, or object density, or both, the dependency of the view volume culling's complexity on n implies that the algorithm would never halt. Ideally, this dependency would be removed to alleviate the impact of view volume culling with increasing size and detail. This scenario, whilst unlikely in practice, could be realized through procedural generation of environments.

A *locale management* system, sometimes considered as *awareness management* [13] [161] can be used to quickly identify a finite volume subset $L \subseteq S$ containing m objects of the scene S to undergo view volume culling. The locale can also define a high resolution coordinate system for the scene region being viewed, which itself is defined in a lower precision coordinate system [161] to support larger scene sizes. The locale management algorithm would ideally be independent of n and have better than linear scalability. For accuracy, the remaining scene objects in $S - L$ would not possibly contribute to the image when using a local illumination model. If $m \neq \infty$, such that the locale has a finite density of objects, such an algorithm may be guaranteed to halt.

If a locale L is defined, view volume culling is still beneficial to stop all of the locale's primitives being sent through the pipeline. View volume culling processes m locale objects to result in a set $V \subseteq L$ of $q \leq m$ objects that either intersect or are contained by the view volume. Testing all m objects for view volume visibility culling would result in a linear $O(m)$ algorithm. Logarithmic scalability can be introduced using spatially efficient, balanced hierarchies. In the theoretical case where $m = \infty$, the view volume culling process may never halt. Ideally, the complexity of the chosen algorithm would be independent of m, but this is not simple to achieve.

Object and surface representations that use fixed levels of detail have speed and scalability issues. Firstly, unnecessary processing is undertaken in surfaces where the projected primitive density is greater than the pixel density in the frame buffer. This projected primitive density increases with viewing distance. Even for low polygon count models, the polygon density can still be greater than is required for the application, or more than is suitable for the system's performance. Fixed level of detail introduces linear $O(p)$ complexity rendering, where p is the number of primitives in the set of potentially visible objects V in the view volume. All primitives of all q potentially visible objects are entered into the pipeline.

In the theoretical case where $p = \infty$ in an infinitely detailed environment with a finite number q of potentially visible objects in the view volume, the dependence on p implies that the rendering algorithm would never halt. This scenario is unlikely to occur in practice, but may be possible through procedural generation of detail.

A level of detail algorithm [66] [100] can be used for a finite number of potentially visible objects $q \neq \infty$ in V, to impose a limit on the resolution of scene detail, resulting in r primitives used to render the image, without dependence on p at render time, assuming pre-computation. With $q \neq \infty$ and $r < p \leq \infty$, the algorithm will halt. Detail limits can be chosen, such that they satisfy performance criteria such as frame rate and consistency [66] [102] [207], in addition to qualitative criteria. This algorithm must only address a finite number of primitives r that represent detail to make the algorithm independent of p and dependent only on the amount of detail that must be introduced. Such algorithms have undergone recent research and are commonly termed *output sensitive*.

The majority of rendering systems render all q objects in the view volume, whether visible or occluded by other objects closer to the viewpoint. The rendering algorithm's complexity is therefore linearly dependent $O(d)$ on the maximum number of surface layers d over distance in the view volume, often termed the *depth complexity*. Again, in the theoretical case where $d = \infty$ (a case where $q = \infty$) the renderer would never halt. Level of detail control that is capable of aggregating multiple surfaces, breaking topology, may also be able to aggregate depth layers, such that the dependency on d may be broken.

Occlusion culling [40] [17] [78] [228] is a solution that avoids the rendering of objects or surface regions in V occluded by those closer to the viewpoint, subject to the camera's projection, resulting in a visible set M. Ideally, the algorithm would be independent of the depth complexity d such that hidden objects are not even tested for visibility, but this is difficult to achieve in practice and occlusion culling algorithm running time typically increases with d. It may be possible for the algorithm to be independent of d, if a rendering termination function can be derived that is capable of establishing when no further depth layers contribute to the image, i.e. that the image is complete.

In the case where $d = 1$, for example with the plan view of a pyramid surface, occlusion culling solutions do not offer any increased performance and may even slow the system down due to their additional analytical overheads of establishing that there is no occlusion. However, the incidence of no occlusion in a scene is uncommon in the real world, though this is also dependent on the size and complexity of the locale.

Standard rendering algorithms do not employ occlusion culling techniques. Therefore, when all primitives in the view volume are rendered, depth ordering is required so that objects further from the viewpoint are correctly obscured by those closer to the viewpoint. When depth ordering of primitives is not solved by the rendering algorithm, z-buffering [63] in the pipeline is usually employed to resolve at pixel level. In this case, primitives can be drawn in arbitrary depth order, although back to front ordering can still be required for correct transparency compositing. This *pixel over-draw* used in z-buffering may be less efficient than algorithms that detect and discard occluded objects or surface regions at lower resolution than that of the pixel. Even if occlusion culling is used, some depth ordering may still be needed to resolve ordering around silhouettes of objects in the image, where primitives are only partially occluded and overlap. Pixel overdraw should be kept to a minimum, if suitable alternative approaches are more efficient.

The distinction between an object and the primitives that represent its geometry, shows a substantial inconsistency. Even though object hierarchies are often used, the surface shapes are usually described at the leaves of the data structures using sets of primitives that differ in nature to the hierarchy. This distinction has enforced a paradigm that is focused on switching scene representation data structures at a particular scene scale. Existing solutions to the problems discussed above, have been derived to make decisions based on object properties or primitives that are only applicable at the scene scales used, rather than in a scale independent way. For example, an occlusion culling algorithm might only be applied to objects or only to primitives. Also, level of detail algorithms typically only operate on surfaces described by primitives [69] [100] and not the object hierarchy, though later work has focused on both [144] [14] [57].

An example problem of object level occlusion culling can be demonstrated, where one object near the viewpoint has a surface that occludes part of another object further from the viewpoint. An object level algorithm may not cull any part of the rear object, resulting in more primitives entering the pipeline and greater rendered depth complexity with increased pixel over-draw. Conversely a primitive level algorithm that only uses primitives at one level of detail, is likely to address all primitives to establish occlusion.

An example problem of a level of detail control system that only applies to primitives and not object hierarchies is also easily demonstrated. The algorithm may provide bene-

fits at one scene scale, but not another. For example, consider a scene with a room containing a small number of highly detailed objects. In this case, the small set of objects to be rendered have sufficient image size with applicable surface based level of detail control to be capable of limiting the amount of detail addressed. In contrast, consider a city scene with a large set objects in view. Each object is likely to be very small in the image. Even if the lowest level of detail surface representations are used for these models, the high number of objects that must be rendered affects the efficiency of the algorithm. In effect, the objects become primitives in a system with no tangible level of detail control at these scene scales. Additional practical problems of temporal aliasing also occur in this case, where a scene can be seen to 'sparkle' because only one of several objects that are rendered to a single pixel is displayed in each image and the object may change between frames. Level of detail techniques can provide anti-aliasing for this situation by representing the contributions of multiple surfaces or objects to the pixel. Conversely, a level of detail system that only operates on hierarchy has no ability to control scene detail at the scale of the room example.

All algorithms using data structures that are inconsistent across scales, may need to be more complex than is necessary, because they are forced to consider the multiple (*ObjectGroup*|*SingleObject*|*Primitive*) geometry and data structures in the representation. This issue will affect algorithms for scene management, rendering, dynamics, analysis, physical simulation and interaction. In the case of rendering, algorithms for locale management, view volume culling, level of detail control and occlusion culling are all affected. Therefore, some form of *homogeneous* scene representation would be appropriate, that is the same across all scene scales.

The problems highlighted in this section are considered directly in the framework for the algorithm's design in Section 3.1.

The next section discusses the scope of this thesis.

1.5 Scope

This section presents a scope for the thesis, to constrain the form of solution such that a result can be practically achieved in the time available.

The algorithm is intended to focus on real-time rendering and only considers local illumination models in favour of performance. Only static scenes will be addressed, whilst not prohibiting further extension to scenes with dynamic surfaces, objects or lighting.

The algorithm will focus on fast, sub-linear scalability rendering of very large, highly detailed environments rather than concentrating on higher quality rendering of single objects. Rendering is considered of primary importance and the applicability of other scene representation oriented tasks such as analysis and simulation are considered out of scope.

Complex image quality metrics and physiological and psychological perception will not be considered.

An assumption is made that image correctness is not necessarily a requirement and that some degree of image artifacts that may arise through approximations, may be considered acceptable if they do not have substantial subjectively judged impact.

1.6 Contributions

This thesis presents a single scale independent scene representation. A new point based rendering algorithm is introduced, offering a solution to the key scalability issues described in Section 1.4. Points are used to describe small surface areas and a *splatting* technique is used to rasterize them in the image at near pixel or pixel sizes.

Moreover, it derives a unified solution to multiple sub-problems to achieve typical sub-linear scalability based on *locale management*, *view volume culling*, *level of detail control* and *occlusion culling*. An approximate *depth ordering* solution is also included. These aspects are tightly integrated, simultaneous in their operation and complimentary in their sharing of sub-tasks and data structures.

To the author's knowledge, it is the first point based rendering technique that derives an occlusion culling solution explicitly tailored for point based rendering, without requiring hardware visibility queries. It is also one of the first to embed a point rendering system into a scene graph architecture with concepts common to conventional graphics APIs such as groups, instancing and inlines. Procedural scene generation is also addressed as a framework for scene creation and detail enhancement with read on demand access of the scene graph from backing store.

In addition, to demonstrate the versatility of the architecture, hierarchical shadow mapping, point-object collision detection and object-object collision detection are also addressed.

1.7 Overview of the Solution

This section gives an overview of the nature and properties of the algorithm presented in this thesis [24]. A detailed description of the theory is given in Chapter 3 and Chapter 4 with implementation details in Chapter 5.

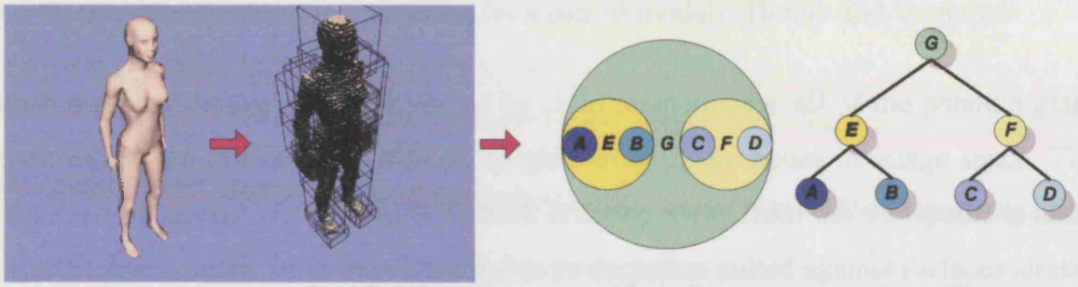
The algorithm is an approximating solution that has applications in real-time and off-line rendering, primarily where image correctness is not a critical requirement. Examples of future use include large scale and architectural scene visualization, games and perhaps may be extended to other domains such as film effects and animation production, with improved accuracy. The system is currently only demonstrated with static scenes, but does not prohibit its extension to dynamic scenes.

A hierarchical point based rendering algorithm is presented that is designed from the outset to achieve the improvements in scalability described in Section 1.4. Direct comparison of the algorithm's properties with those of standard rendering algorithms is not straight forward, due to the lack of discrete objects in the scene data structure and continuation of the data structures to surface level descriptions. However, the concept of either an object or a primitive will be interpreted as equivalent to a point.

A hierarchical scene representation termed the *scene tree,* provides a multi-resolution binary tree of *scene nodes* for level of detail control (see Figure 3). The leaf scene nodes are sourced by sampling a given scene. The hierarchy is built using a BSP tree like spatial partitioning process. Each branching scene node is a geometrical and attribute approximation of all leaf scene nodes in its sub-tree. The scene tree also provides a homogeneous, scale independent scene representation upon which to base rendering and dynamics algorithms. The leaf scene nodes (shown as A, B, C, D) are sourced from high resolution polygon models using a voxelization technique [118] [47] [105], but this data set is not structured, so any source that provides contiguous surface samples can be used.

The level of detail control technique contributes to improved practical speed and scalability by introducing a multi-resolution representation that supports multi-resolution view volume culling, back face culling, occlusion culling and depth ordering, in addition to simply limiting the amount of detail finally rendered in the image.

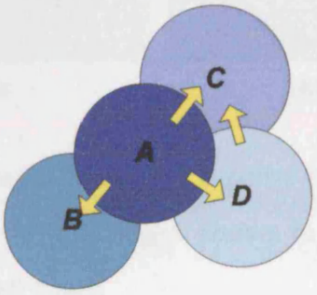
FIGURE 3. **Voxelization to scene nodes and scene tree hierarchy construction**



The scene tree lends itself readily to a form of *locale management* that can typically be carried out independently of the number of points in the scene j, incrementally identified as a suitable root node in the scene tree, resulting in m points in the locale.

An image is rendered by refining from the locale root node, down the scene tree, successively evaluating and replacing nodes with their two child nodes as higher level of detail approximations. When the level of detail is considered sufficient for rendering tasks, refinement can cease. Parts of the scene tree can be culled away during traversal, by view volume, back face or occlusion culling, shown in Figure 7.

FIGURE 4. **Simple image graph example with image nodes** *A, B, C, D*



A key data structure termed an *image graph* incrementally maintains a description of the active nodes used during scene tree refinement and their relationships in object and image space. The image graph provides information for depth ordering and occlusion evaluation. Figure 4 shows an example of a full refinement of the scene tree in Figure 3. When a scene node in the scene tree is refined, its *image node* in the image graph is also refined to two corresponding children. Culled nodes are removed from the image graph. Rather than a 'scene tree' refinement, we can consider it a process of 'image graph'

refinement. Figure 5 shows an example of a refined image graph with (a) image nodes and (b) relations between image nodes for a pair of models (Bunny and Venus).

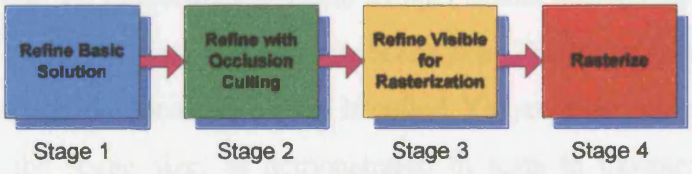
When a refined image node is replaced by child image nodes, all of the parent's graph relationships are re-evaluated with all neighbouring image nodes in image space. This refinement is carried out in a front to back ordering away from the viewpoint to allow scene surfaces further from the viewpoint to be occlusion culled against surfaces already established to be nearer the viewpoint. Refinement of image nodes terminates in one of several cases. The first case is simply if the image space level of detail criteria are met for an image node. The remaining termination cases are due to view volume culling, back face culling and occlusion culling respectively.

FIGURE 5. **Image graph nodes and relations (Bunny & Venus)**

(a) Image graph nodes **(b) Image graph relations**

Rather than refine in a single stage, due to the expense of image graph based occlusion culling, refinement is split in to several stages, shown in Figure 6, that operate between specified levels of detail in image space.

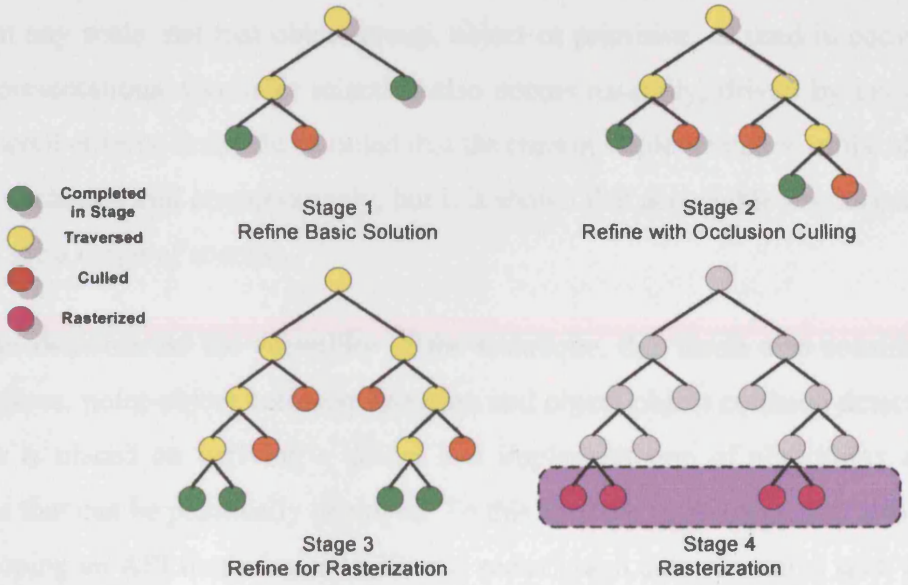
FIGURE 6. **Rendering stages**



An example of refinement in each of these stages is shown in Figure 7. Stage 1 simply refines the image graph, adding scene detail without occlusion culling to set up a basic image solution as quickly as possible. Stage 2 refines the image graph further, with

occlusion culling to a specific level of detail in image space, to establish a potentially visible set. The third stage quickly refines the potentially visible set of nodes in the image graph depth first, without image graph refinement, to a higher level of detail for the final rasterization. Stage 4 then splats the results of Stage 3 using the standard pipeline described in Section 1.2.

FIGURE 7. Scene tree refinement for stages 1 to 3 and rasterization stage 4



The example in Figure 7 shows nodes that are *completed*, *traversed*, *culled* and *rasterized* in each stage. The result of each stage is passed to the next, with *completed* nodes continuing to undergo refinement. In practice, a real scene tree has substantially more levels. Note that only addressed scene tree nodes are shown. This form of addressed sub-scene tree is also maintained in the system between frames for simple caching, termed an *image tree* of image nodes that are also present in the image graph.

It will be shown that the rendering algorithm's upper bound complexity is $O(j^2)$ where j is the number of leaf level scene nodes in the scene tree. The lower bound complexity is $\Omega(1)$ in the case where the locale root node is culled. The average complexity is likely to be sub-linear in the scene size, as demonstrated in tests in Chapter 6. Importantly, because level of detail control imposes an upper limit on the refinement process where only r scene nodes where $r \leq j$ are required to rasterize the image, all refinement based

processes can be considered as having an upper bound $O(r^2)$, independent of the highest levels of detail available, realizing an *output sensitive* solution.

The occlusion culling technique is not based on any concept of discrete objects or surfaces. Therefore, the algorithm naturally considers *occluder fusion* [40] [179] where surfaces that are spatially separate in object space, but overlapping in image space, are considered as a combined occluding body. Occlusion testing and culling functions can be applied at any scale, not just object group, object or primitive, as used in conventional scene representations. Occluder selection also occurs naturally, driven by image space level of detail criteria. It should be noted that the current implementation of the algorithm does not occlusion cull conservatively, but it is shown that acceptable results can still be achieved for a range of scenes.

To further demonstrate the versatility of the technique, this thesis also considers real-time shadows, point-object collision detection and object-object collision detection. An emphasis is placed on deriving a design and implementation of algorithms and data structures that can be practically deployed. To this end, the implementation also focuses on developing an API that mimics traditional scene graph data structures such as group nodes, instancing and inlines, for scene control, rather than surface representation. Implementation issues of procedural generation, geometry and scene graph compression, architecture extensibility and speedups are also examined in depth.

Tests with scenes conventionally consisting of several hundred billion polygons have been shown in Chapter 6 to render at interactive frame rates, demonstrating typical sub-linear scalability for the scenes used.

1.8 Overview of this Thesis

Each chapter begins with a one page overview of the chapter's structure and ends with a summary of the main points.

Chapter 2 gives background information that covers standard rendering optimizations such as locale management, level of detail control, image based rendering and a range of culling techniques. The chapter ends with an examination of point based rendering.

Chapter 3 then moves on to describe the *Canopy* point based rendering algorithm at a high level, which is then examined further in subsequent chapters.

Chapter 4 then discusses the addition of hierarchical shadow mapping and collision detection based on existing data structures of the rendering architecture.

Chapter 5 discusses aspects introduced in Chapter 3 with more implementation detail, giving maths, pseudo code and discussing problems in depth.

Chapter 6 tests aspects of the system's correctness, qualities and scalability, giving details of the systems performance on its own and in comparison to standard polygon rendering. The end of this chapter then compares the results to a select number of other systems in the literature.

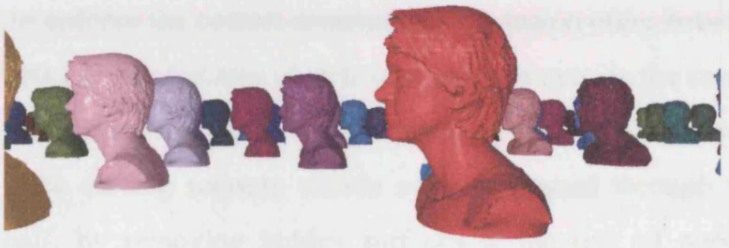
Chapter 7 derives conclusions, strengths and weaknesses and gives a range of topics for future work.

A glossary of both standard terms and terms introduced in the thesis is then given.

The appendix towards the end of the thesis overviews the system's architecture, implementation issues, the SCT file format and further results of tests in Chapter 6.

Finally, the references detail citations given throughout the thesis.

Chapter 2 Background



This chapter describes a range of rendering optimizations chosen or applicable for incorporation in to the Canopy system and examines previous work in each area. A summary can be found at the end of this chapter, that discusses the algorithm in relation to many of the aspects covered in this chapter.

To begin, each type of ***rendering optimization*** is overviewed in Section 2.1.

Locale management is then examined in Section 2.2, ***level of detail control*** in Section 2.3, and ***visibility ordering*** in Section 2.4.

Hierarchical culling techniques including ***view volume*** and ***back face culling*** are discussed in Section 2.5 and Section 2.6. ***Occlusion culling*** is covered in Section 2.7.

Image based rendering as an optimization technique is then discussed in Section 2.8.

Previous work specifically relating to ***point based rendering*** is reviewed in Section 2.9, followed by a ***summary*** in Section 2.10.

2.1 Forms of Rendering Optimization

Several forms of rendering algorithm optimization will be addressed in this chapter, as outlined in the framework for the solution given in Section 1.4. Often, each solution type is addressed separately in systems, without any intentional integration.

To recap, ***Locale management*** is the process of reducing the scene to one or more subsets that are addressed for rendering. ***Level of detail*** techniques provide multi-resolution models or surfaces for use with a level of detail control system to select appropriate resolutions subject to the system's resources and application requirements. ***Visibility ordering*** is the process of controlling the rendering order of objects or primitives such that either a back to front or front to back depth ordering can be achieved, as required by the rendering algorithm for optimization purposes such as occlusion culling, compositing transparency or to enforce the correct accumulated occlusion order between opaque surfaces. ***View volume culling*** discards objects or primitives outside the camera's view volume to reduce the number of objects or primitives passing further down the rendering pipeline. ***Back face culling*** reduces visible surfaces passed through the pipeline by approximately half, by removing hidden surfaces at the rear of polyhedral models. ***Occlusion culling*** removes objects or primitives that are not visible because they are hidden behind others in the scene. ***Image Based Rendering*** is not directly used by the algorithm in Chapter 3, but it is related and applicable.

Each of these topics is examined in the coming sections, addressing solutions for standard polygon rendering as well as point based rendering. Each section will briefly state the problem and overview a number of existing solutions in the literature.

Point Based Rendering is examined in depth at the end of this chapter.

2.2 Locale Management

2.2.1 The Problem

Rendering algorithms generally take a number of successive steps to reduce the size of the data set addressed, using a range of techniques. One of the most top level techniques is *locale management* that takes potentially fast decisions to reduce a very large scene S to a subset $L \subset S$, where only the data in L is addressed for rendering. Ideally, no data in $S - L$ is addressed. Locales may be calculated at runtime, but are often specified off-line, but identified as required at runtime. The finite nature of fixed and floating point representations may constrain the maximum size and resolution of scenes. In addition, designers may wish to create different regions of a scene independently and combine them for use. Locale management aids in these areas.

2.2.2 Locale Management Techniques

Locale management is largely overlooked in the literature, but a limited number of examples have been developed. A locale management system can either attempt to work with the scene data structures already dictated by the rendering architecture, or modify, add or apply scene design and data structure constraints to suit their function.

The locale subset may be dramatically smaller than S and potentially take very little processing to identify. A prime example is a flight simulator, where the whole planet is represented, but only a subset surrounding the aircraft is immediately required, with a locale size that will encompass detail to the horizon, but not necessarily beyond.

Scene dynamics within L may dictate that L be recalculated, with the potential fetching of required scene data. In many applications, user motion is constrained by a navigation type, such as walking or flying. Knowledge of the temporal bounds of the navigation type can be used when designing locale management systems. It may even be, that the locale management system dictates navigation constraints, such as maximum velocity.

Locales allow scene regions to be designed separately and combined for use. One particular problem that locales can solve is the limited numerical precision of fixed or floating point representations that limit the size and resolution of a representable scene. For

example, values in one scale and resolution can position a locale, while another scale and resolution is used for the local region's description [193]. At times, locale management algorithm classification may overlap with other forms of scene reduction, such as occlusion culling. One such example is the *cells* and *portals* concept used by Teller [200] and Luebke and Georges [137], where scenes are designed as connected celled regions such as rooms that are accessed via doors, windows or other geometric interfaces of arbitrary shape and limited size. Knowledge of their connectivity and visibility from the viewpoint is used to identify cells and cull away substantial regions in far cells by clipping to the boundaries of their geometric interfaces. This defines sequences of required cells and may define an exact visible set or a less accurate, but usually conservative potentially visible set (PVS) of primitives. Visibility may pass through multiple portals to identify multiple cells that must be included in the locale for the image. The cell and portal concept generally imposes cells and portals to be explicitly identified in the scene's design and is not suitable for all scene types, particularly large, open environments with low occlusion.

Most applications are single user, but multi-user collaborative virtual environments (CVEs) need techniques that scale to large environments, interactive changes and large numbers of users. Locales can be useful in CVEs to localize and restrict the scene state and communication [13] [161].

Locale selection policies can be based not only on geometric qualities, but other aspects of an environment that the user may focus on, using *awareness* policies, such as in the Massive-3 system [161].

Procedural scene generation algorithms may also potentially benefit from locale management systems, where differing algorithms could be invoked based on user position and required scene scales. Notably, the use of procedurally generated locales makes the concept of infinitely sized and detailed universes practically achievable, although some issues remain, such as the requirement for an infinitely sized viewpoint location variable.

The locale management discussed for use the Canopy system in Chapter 3 is basic, initially intended to quickly cull very large scenes, leaving just the surrounding region within a specified radius of a single user viewpoint, for example, up to the horizon.

2.3 Level of Detail Control

2.3.1 The Problem

For classical level of detail (LOD) control, an analogy can be drawn with the task faced by an artist attempting to paint a reasonably realistic picture of a scene. There are finite resources, in this case time, a potentially large canvass to be filled and possibly restrictions on paints, some of which could be very expensive. A style evolved primarily in Paris in the late 19'th century known as *impressionism* and more specific to this thesis, *pointilism.* A common characteristic of the techniques were an attempt to capture visual realism objectively, using lighting observations and colour usage in an effective but efficient way. Artists like Renoir and Monet used impressionism to do just that - give an impression without explicit detail. In a way that is rather more manual, these artists came up against and solved much the same problem that can arise in computer graphics. Although the images are not always photo realistic, they exist to serve a purpose in their own right and result from an effective use of resources.

Level of detail control has been originally intended as a method for achieving higher frame rates with a higher degree of frame rate consistency by solving the problem of removing detail from models that was considered to be of less visual value, whether the changes were visible or not. The reduced set of primitives may render more quickly through the pipeline, particularly if the system is limited by pipeline vertex processing. If the system is fill rate limited, level of detail may have far less impact because the screen space area is usually similar, even at low levels of detail.

A level of detail control system is required to make decisions on how to distribute detail. Rendered without occlusion culling, the control system must choose how to distribute detail even to occluded objects. If occlusion culling is used during rendering, the problem is redefined in terms of distributing detail over the visible surfaces in image space and any conservative over-draw regions that may remain. In the case of the artist, nature has done the occlusion culling for them. If the performance of the hardware platform is high enough and there is sufficient geometry defined, the level of detail problem is simply one of limiting the maximum detail to pixel size, particularly for reasons of not surpassing the Nyquist frequency of the image sampling that would cause aliasing. At times

when hardware is not sufficiently quick to identify and render primitives at pixel levels of detail, the system must choose how to distribute detail over the set of visible objects or surfaces. In the case where geometry is simply not defined, lower level of detail approximations may suffice. In the case of polygon rendering, large polygons are used as a piece wise linear interpolation of the surface samples. In point based rendering (PBR), the *splatting* technique is often used to fill under-sampled image space regions. However, in this thesis, we will take the approach that geometry should be sufficiently defined to maximum levels of detail required for the application and that in cases where source data level of detail is too low, it is preferable to incorporate a framework for increasing detail in object space, rather than attempting to approximate under sampling in image space.

Although this section covers a wider range of geometry, decimation and LOD control techniques, most are applicable or could be interpreted to point based systems. The reader is also referred to Section 2.9.4 for additional information on specific point based approaches to LOD.

This section overviews model calculation and control techniques for both polygon and point based systems. Several surveys are also available [55], [67], [95] and [166].

2.3.2 Level of Detail Geometric Techniques

Architectures incorporating level of detail need to provide multi-resolution models or surfaces and incorporate a control system for selecting which levels of detail to use given the current scene state and viewpoint. This section will examine a range of techniques given in the literature, primarily since the mid 1990s. Control systems are examined in Section 2.3.5. The reader is also referred to Section 2.9.4 for a specific survey of sampling and data structures for point based systems and Section 2.7 where combined LOD and occlusion culling systems are examined.

Geometric model approximation techniques generally operate either on whole models or local regions. The former case can be controlled by selecting a level of detail for each model in the scene, while the latter selects surface regions to increase or decrease detail as required, using a technique referred to as *viewpoint dependent selective refinement*

[102]. Level of detail systems that provide a small sequence of levels of detail are generally termed *discrete*, while systems that provide a large sequence of levels of detail are termed *continuous*, offering greater flexibility. LOD systems have concentrated on the continuous form, particularly the Progressive Mesh method introduced by Hughes Hoppe in [99] [100], and others including Lau and Green [126], Schmalsteig and Schaufler in [181] and Garland and Heckbert in [69]. Many systems that produce a discrete set of models, are capable of producing a large number of them if required and as such are categorized as continuous. A truly continuous method is that of Progressive Meshes [100] that offers a *geomorph* technique to smoothly linearly blend one LOD through to another by adding or removing the required triangles and blending linearly between their positions in the two resolutions.

Representing a whole model using a single resolution incurs a problem in that surface regions close to the viewpoint may require greater detail than further regions. This implies a requirement for non-uniform LOD within a model, to achieve viewpoint dependent selective refinement. Other regions, such as curved areas of the object's silhouette may also require more attention than interior regions. This form of system requires the geometry and control systems to be tightly coupled at design time and may require greater levels of analysis and geometric operations at runtime. Examples of polygon based systems include additions made to the Progressive Mesh system by Hoppe [100] [102], Schmalstieg and Schaufler [181], Hoppe [101], Lau and Green [126], Luebke and Erikson [136] and Xia and Varshney [221].

Approximation error metrics are generally based on object space Hausdorff distance [33] minimization between the original and approximated surfaces, an explicit example being the *simplification envelope* by Cohen and colleagues [39]. In viewpoint dependent selective refinement, image space error metrics are also used to identify regions of change.

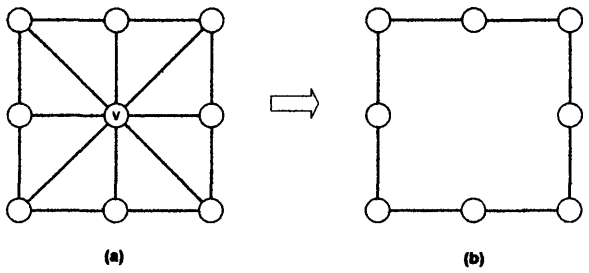
If topology is preserved, as in methods like the *progressive mesh* [100], model surfaces will never merge, limiting the lowest LODs possible. This is particularly restrictive in large scenes, where distant viewpoints may benefit from high mesh decimation levels, that may involve breaking topology to blend surfaces and models to an alternate topology. Examples that do not preserve topology include Garland [69], Luebke and Erikson [136] and GAPS by Erikson and Manocha [56].

The space of possible approximations of an original mesh M is infinite. Constraining the solutions to the removal of a single geometric element in an iterative way, still yields a huge factorial space which can not be bounded by a polynomial time algorithm. Therefore, most methods resort to the use of a *greedy* optimization algorithm, an introduction to which can be found in [164], that accepts local optima. Typically, at each stage, the previous resolution is modified by applying a single simplification operation. The three components of surfaces vertices, edges and triangles or other polygons can each be removed from a mesh. Various methods have incorporated the removal of one or more of these types. Methods resulting in the smallest changes to surfaces generally offer greater control. All surface operations must usually be invertible to allow refinement of detail through the re-introduction of geometry, if discrete models are not used.

All techniques at their most intrinsic level are based on the removal of vertices and potentially the re-introduction of vertices to approximate those removed. Explicit examples of this are the quantization groupings of Rossignac and Borrel [170] and the Octree [109] based grouping criteria of Schmalstieg [180].

In polygon meshes, the removal of a vertex results in the invalidation of all polygons defined by the vertex. If the vertex was connected to n triangles, for example, n triangles will be removed. This creates a hole which will have to be re-triangulated, as shown in Figure 8. A re-triangulation can reuse an existing vertex, equivalent to an edge collapse.

FIGURE 8. Impact of single removal of vertex V invalidates all connected triangles

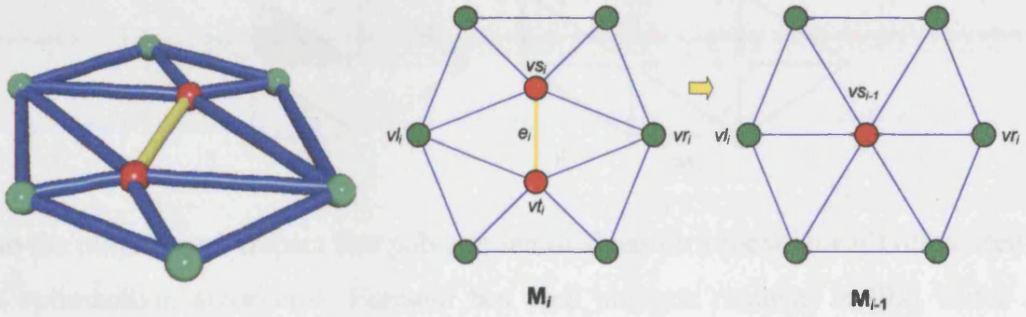


Edge removal removes polygons defined by the edge. In a triangle mesh, the two wing triangles will be removed. A simple re-triangulation can merge the two edge vertices, known as an *edge collapse* operation, used in the Progressive Mesh method [100] [102].

It is the simplest and most powerful alteration for a triangulation as it removes only two triangles and is an implied re-triangulation of the case in Figure 8.

An example surface area suitable for an edge collapse operation is shown in Figure 9. Edges are marked in blue. The central edge is suitable as it is not connected to more than two triangles. The collapse of edge e_i defined by vertices vs_i and vt_i forms a new vertex vs_{i-1}. The position of the vertex is not the same for different methods. Simple placements of vs_{i-1} at vs_i or vt_i can be performed. Alternatively, $vs_{i-1} = (vs_i + vt_i)/2$ can be used. These are unlikely to be optimal replacements with respect to the approximated surface.

FIGURE 9. Edge collapse region and edge collapse from resolution i to $i-1$



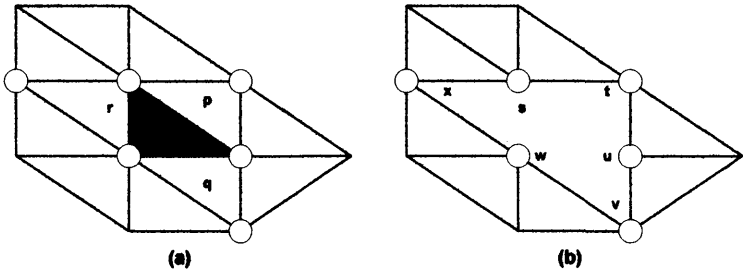
The progressive mesh method terms the inverse of an edge collapse operation a *vertex split*. The examples in Figure 9 show a simple polygon corona surrounding the original interior vertices vs_i and vt_i. In reality, there may be more than five polygons defined by these vertices. This does not complicate the method, as the labelled vertices and edge are still the only factors which affect the geometric alteration. Sander and colleagues [174] form a texture parameterization to reduce distortion for texture mapping.

Popovic and Hoppe extend the Progressive Mesh to a Progressive Simplicial Complex [160] that allows topological changes. A non topology preserving form merges two vertices which are not necessarily connected to an edge, as in Garland [69] who uses a least squares measurement to position vertices, Luebke and Erikson [136], Schmalstieg and Schaufler [181] and Xia and Varshney [221].

Surface attributes can also be approximated, such as colour and surface normal. To permit discontinuities over edges, each polygon must store separate attributes for each vertex and must be approximated with respect to discontinuity thresholds.

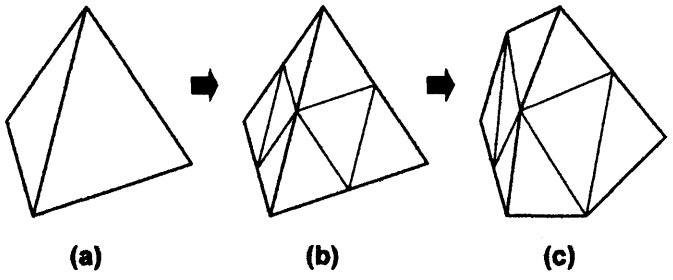
Polygon removal is equivalent to the removal of a polygon's edges. Figure 10(a) shows a triangulation with a central shaded triangle removed in (b) using edge removal, leaving a hole. Triangles p, q and r are also removed due to the removal of one of their edges. An alternative approach that re-triangulates is to collapse a single triangle edge or all edges, equivalent to two consecutive edge collapses.

FIGURE 10. Triangle collapse. Shaded triangle in (a) is removed in (b).



Due to the more severe impact that polygon removal has on a mesh, it isn't often used in mesh optimization algorithms. Hamann has used polygon removal in [91] where all remaining vertices are collapsed to a single vertex, chosen based on a set of heuristics, some of which involve a last squares approximation of local vertices.

FIGURE 11. Tetrahedral subdivision. (a) is subdivided to (b) and modulated at (c)



Subdivisions of surfaces can be used to increase supporting triangles for additional detail. This particular method in Figure 11 shows a tetrahedral subdivision for multi-resolution representation of polyhedra used in a method based on 3D wavelet transforms by DeRose and colleagues [50] [53].

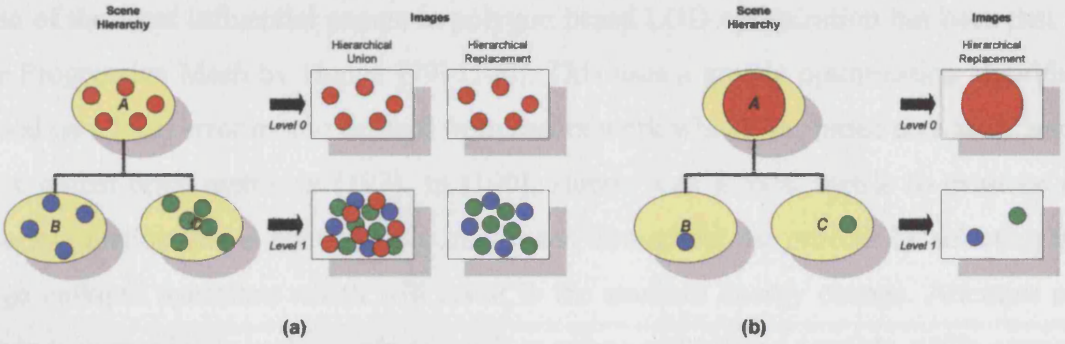
A smaller number of polygon based systems explicitly incorporate hierarchical concepts of level of detail for multiple models. One of the first and conceptually comprehensive was Clark [35] who used bounding sphere hierarchies with interior nodes approximating child nodes, to support level of detail, view volume culling and occlusion culling and utilizes frame coherence. The HLOD system by Erikson, Manocha and Baxter [57] uses a number of discrete LODs in all branch and leaf nodes of a hierarchy, using spatial partitioning to separate large objects. A branching node's LODs geometrically represent LODs of the union of all geometry in the branching node's child graph. Leaf nodes have a regular range of discrete LODs. Their control system has modes for quality and frame rate targets. A high degree of frame rate consistency is demonstrated for large models, with a target frame rate. The authors also apply techniques to support dynamic scenes with a degree of tolerance when motion is small and hierarchical rebuilds when the hierarchy's grouping becomes less efficient.

Point based systems represent surfaces using points or primitives centered at points. Many point based rendering systems use hierarchies of points, either taking the union of points down the hierarchy to increase the surface sampling density, or replacing point samples with a larger number at a higher resolution for increasing levels of detail. An example is shown in Figure 12(a) with a hierarchy of height one. Images for union or replacement are shown in the right columns for cases where refinement is just to node A at level zero or has traversed down to include both child nodes B and C at level one. Each type requires A to represent B and C, where in the replacement scheme, nodes in A may be required to bound child nodes in B and C, shown in a binary branching example in Figure 12(b). Point set union is an approach more unique to point based systems. Replacement on the other hand, is often very similar to the existing approaches described in this section for polygon based LOD systems, where simplification operations can often be described in terms of point merges or edge collapses as the most atomic simplification operation. Point union can have advantages over point replacement due to reuse of data that is also more suitable for cached Graphics Processing Unit (GPU) implementations, e.g. in the Randomized Z-Buffer [205] and Layered Point Clouds [72].

Point merging options can be described using trees, where earlier polygon based examples include Hoppe's adaptation of progressive meshes to view dependent selective

refinement [102] or Schmalstieg and Schaufler's cluster tree [181]. These techniques are just as applicable to point based systems, regardless of any need or technique for the interpolation of surface samples. Common partitioning or clustering techniques used in point based systems include *octrees* [109], *k-d trees* [15] and *principle components analysis* (PCA) [112].

FIGURE 12. Hierarchical point union or replacement



An approach to level of detail in point based systems is simply to sample the highest level of detail geometry, with increasingly higher density. This is carried out by hierarchical sample accumulation through point set unions and more explicitly in realtime systems such as the Randomized Z-Buffer by Wand et al. [205] and Layered Point Clouds by Gobbetti and Marton [72].

Procedural generation systems also offer great potential for multi-resolution detail, where instead of just removing detail, existing detail may be enhanced, for example by interpolating control points, instancing new local detail based on context, or explicitly generating new surfaces from functions.

The reader is referred to Section 2.9.4 for details on multi-resolution geometric sampling and data structures used in point based systems for level of detail, including procedural detail enhancement systems.

2.3.3 Error Metrics for Optimization

Level of detail error metrics are required to evaluate the accuracy of an approximation. This may be required globally or locally in object space and also in image space for LOD control. A global system will evaluate an approximate model relative to the original,

such as in the Metro system [33] by Cignoni and colleagues. Local metrics measure smaller successive changes to models, usually based on greedy approximation algorithms that accept local optima in terms of the global hausdorff distance from the original surfaces, initially preserving curved regions in systems such as Hamman [91], in preference to flatter regions. They may also measure surface attributes and preserve sharp discontinuous features such as hard geometrical edges and well defined colour changes.

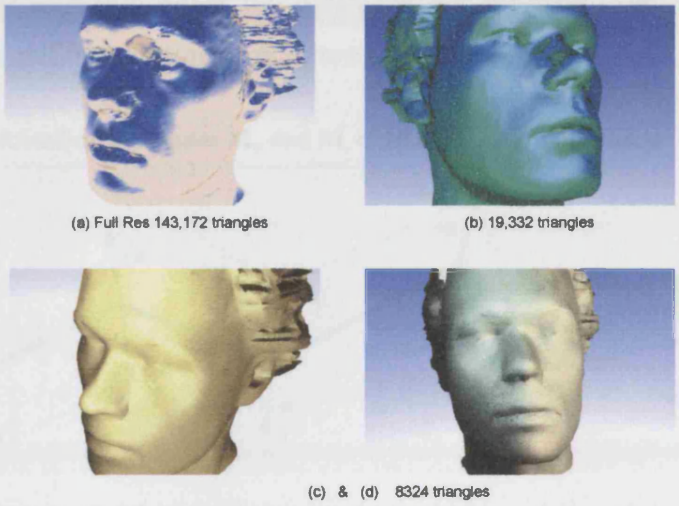
One of the most influential papers in polygon based LOD optimization has been that of the Progressive Mesh by Hoppe [99] [100]. This uses a greedy optimization algorithm based on a local error metric derived from earlier work which was based on a more complex global error metric in [103]. In [100], Hoppe uses a local metric to evaluate an *energy* function that is continuously minimized throughout the process by selecting the edge collapse operation which will result in the smallest energy change. Attempts are made to approximate scalar attributes such as colour and surface normals, while preserving discontinuity curves resulting from sharp changes in both geometry and surface attributes. The surface can be applied to 2-manifold regions only. Each legal edge collapse is applied to the energy function and maintained in a priority queue. On each iteration, the edge collapse with the smallest energy function is removed from the priority queue and collapsed, resulting in a new mesh. Figure 9 shows the geometry of the collapse operation. More formally, the error metric applied to the edge collapse operation is stated as follows:

$$E_K = min[E_{dist}(V) + E_{spring}(V) + E_{scalar}(V, S) + E_{disc}(V)] \quad \text{(EQ 1)}$$

where $E_{dist}(V)$ is a squared distance metric of the approximated vertex from vertices in the mesh, with the spring energy term $E_{spring}(V)$ penalising long edges. The $E_{scalar}(V, S)$ and $E_{disc}(V)$ preserve scalar and discontinuity surfaces respectively. The mesh connectivity is represented by K, V are the vertices and S are the scalar attributes. The results appear to be high quality approximations. Figure 13 shows results of Lee Bull's head scan using a modified progressive mesh algorithm that measures tetrahedral volumes caused by edge collapses as an explicit greedy minimization of the Hausdorff distance [33]. The features removed tend to be high frequency low amplitude, forming a low pass filter. The original scan (a) with 143,172 triangles is reduced to 19,322 and 8,324 respectively.

For purposes of comparison, *better than random* metrics were also tested. Edges are selected randomly from the set of legally collapsable edges, which is why the method is better than random. The system very quickly generates a noisy cloud of polygons.

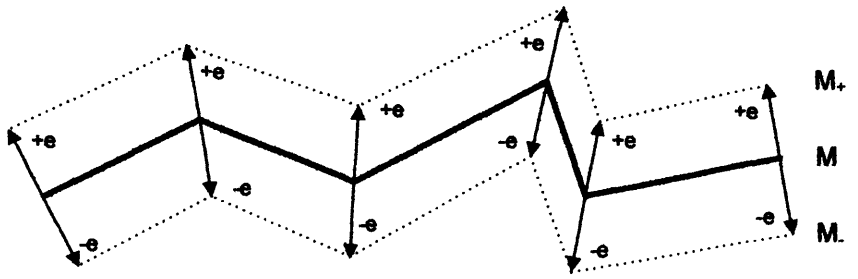
FIGURE 13. Rendered model (a) using Progressive Meshes resulting in (b) & (c,d)



A metric which has received substantial interest in the literature is the *quadric error* method given by Garland in [69]. The method is later extended to handle colour and texture in [68]. The quadric error metric is an elegant, incremental algorithm that selects pairs of vertices to merge, which are not necessarily connected by an edge, so the technique can aggregate locally within the object by discarding topology. A pair of vertices v_1 and v_2 are selected for merging iff they define an edge of the mesh, or $\|v_1 - v_2\| < t$ where t is a threshold. This also allows the algorithm to be constrained to edge collapses if $t = 0$. Each vertex is associated with a set of planes which it is locally responsible for approximating using its connected polygons in lower resolutions. These planes are embodied in a quadratic error metric maintained for each vertex. When two vertices are merged, it is necessary for the new vertex to inherit the error metrics of the two child vertices. This would suggest taking the union of the planes that their local polygonal regions approximate. This is both an elegant and quickly calculable error metric and is shown by the authors to give reasonable visual results. The quadric error metric has also been used in the GAPS system by Erikson and Manocha [56] in a topology breaking simplification algorithm that bridges surfaces, resulting in a reported 3.5 fold average speedup in rendering.

A metric which imposes a more explicit error bound is given in Simplification Envelopes by Cohen and co-workers in [39]. To constrain the simplification process, two geometrical inner and outer surface envelopes are created around the original surface to contain it as shown in Figure 14. To create the envelopes, the original surface is projected in a positive and negative direction along the mean surface normals. The surfaces are constrained to be non self intersecting to preserve their topology.

FIGURE 14. Simplification Envelopes M_+ and M_- of M with error tolerance e



Optimization of the mesh M is carried out within this envelope. If the lower resolution model is constrained to fit within the envelopes M_+ and M_-, it will be within a distance error tolerance e of the original surface. Self intersection conflicts, e.g. in concave regions are solved by reducing e. This has a secondary benefit of reduce the tolerance at curved regions of the mesh, enforcing a finer triangulation than in flat regions. The envelopes are offset surfaces. This isn't the highest quality of offset surface which could be generated as it is not re-triangulated for regions of curvature. However, it appears to serve the purpose. Two mesh optimization algorithms are given in [39]. Both methods are based on creating holes in the surface and re-triangulating them such that they contain fewer polygons. As both methods select a subset of existing vertices in the original mesh, all vertices are guaranteed to lie between the envelope surfaces. A validity test is defined for candidate polygons in the new approximation surface as two separate tests. Firstly, an intersection test is performed for each polygon with both envelope surfaces. Secondly, a self intersection test is performed for the approximation surface to preserve topology. The two methods described differ in that one makes local, small holes in the mesh, while the other makes larger holes based on global information. Both methods are performed off-line and take in the order of about ten seconds to optimize a mesh of 70 thousand triangles on the hardware used.

One of the biggest departures from conventional methods is given in [50] and [53] by Tony DeRose and co-workers. Wavelets[1] are generally a filter method somewhat similar to *short time* (windowed) Fourier transforms used for *Multi-Resolution Analysis* (MRA), but which include an arbitrary comparison function termed a *mother wavelet*. An introduction to wavelets can be found in [195]. In the context of LOD, they are used as a low pass filter acting on the surfaces of a tetrahedron. Figure 11 shows the geometry of this process where tetrahedron (a) is subdivided to (b) and all vertices in the model are modulated to form (c). High pass data is stored as coefficients of a complement space basis. Low pass results are stored as coefficients of another basis, such that the combination of the modulated basis and modulated complement space functions reconstitute the original function. This process is performed iteratively. When refined further, the object will tend towards the original limiting surface. Each triangle on the surface is a least squares approximation of those triangular surfaces it covers in the next highest resolution. The restriction imposed by the subdivision surface is not very practical. To solve this problem, the authors provide a method for approximating a given object using the subdivision topology necessary in [53].

Quadric error metrics, used by Garland and Heckbert [69] are also used in point based systems by Pauly, Gross and Kobbelt [154] and Pauly et al. [156]. Least squares methods are also used by Botsch, Spernat and Kobbelt [21] to define new splats. The Moving Least Squares (MLS) method by Levin [128] has also been widely used for surface approximation, including Alexa and colleagues [3] Fleishman and colleagues [62] and Pauly and colleagues [156].

Point based systems that use point hierarchies with point union or replacement are typically intended for selective refinement control systems, but still require some degree of accuracy in their approximations. Point union systems typically measure the density of sampling from the original surfaces, where as tightness of bound is more applicable to point replacement schemes.

Higher quality point based LOD approaches generally involve down sampling scanned point sets. Wu and Kobbelt [219] use a global error tolerance based optimization system

1. Originally a French concept, they are termed Ondelettes in France

similar to a surface reconstruction and claim fewer surface elements (surfels) are required than alternative approaches for the same error tolerance. Using a seed approach, a surfel is grown to incorporate a local neighbourhood of points using a least squares plane based error metric. This phase ends when the local neighbourhood points are further than a threshold from the plane. An active subset of surfels is chosen that covers the surface using a greedy selection policy. A global relaxation stage then moves surfels on the surface based on repulsion forces. Examples show the Stanford Bunny model with 35k points reduced to 1,371 in 14s.

Pajarola [152] generates a point octree using bounding ellipsoids. The Progressive Splatting system by Wu, Zhang and Kobbelt [220] is a point based decimation system similar to the progressive mesh edge collapse decimation by Hoppe [99] [100] using a greedy optimization (see Section 2.9.4.3). A resampling system is also given by Fleischman and colleagues [62] based on moving least squares (MLS) projection operators for multi-resolution compressed representations.

Pauly, Gross and Kobbelt [154] discuss three methods for point cloud simplification in object space. The first is based on hierarchical clustering methods that group to a single point. The second simplifies pairs based on quadric error metrics. The third is more unusual, simplifying by distributing a required number of samples over a surface using repelling forces.

2.3.4 Progressive Model Transmission and Out of Core Rendering

Multi-resolution models are generally incremental refinements or coarsenings of the current representation of a mesh. The amount of information representing the *delta* between two resolutions can be quite small, depending on the impact of the change. For reasons of working set management, opportunities exist for this data to be streamed, either from memory, hard disk or over networks to provide the viewer with a visualization early on. This visualization may be of the required quality immediately, for example in the case of selective refinement algorithms if the data can be provided quickly enough. Currently, this situation is unlikely particularly over networks and a more common result will be a sequence of changes over time as the user interacts with a system. Deltas between models can also undergo compression to reduce bandwidths required. Hoppe has demon-

strated streaming for Progressive Meshes in [100] as have others [50] [181] and for point based rendering for offline LOD and rendering [6] [72] [98] [172].

2.3.5 Level of Detail Controllers

Having looked at some of the methods used for multi-resolution model representation, this section will examine another important aspect, that of how object or surface resolutions are chosen for rendering at runtime. Surprisingly, there has been less work in this area than in geometric optimization methods. This section overviews a number of control systems that use static LODs and those using viewpoint dependent selective refinement.

The LOD control system can attempt to achieve target frame rates, frame rate consistency or visual quality and typically trade off between these characteristics. The controller takes a set of inputs and must derive a solution to meet a set of targets and constraints as outputs in a stable, efficient, consistent way. Controllers are generally *reactive* or *predictive* systems. A reactive method makes changes to the rendering configurations of future frames based on the system's performance in previous frames. This contrasts with a predictive system which analyses aspects of the scene to be rendered from a particular viewpoint, such as the number of objects and their sizes and positions to state parameters with which to render the next frame. Reactive systems will be subject to substantial inaccuracy under rapidly changing views [66], whereas predictive methods make judgements to solve for each coming frame. Reactive methods have the property of being simple to design, quick to implement and fast to execute. It may be that a predictive method requires substantially more processing time. If it is based on frame coherence, this time may also be inconsistent which may itself reduce the frame rate coherence which the algorithm exists to provide.

Early contributors Funkhouser and Sequin [66] defined the LOD control problem in relation to the continuous *multiple choice knapsack problem* (MCKP). A detailed study is given by Martello and Toth [143]. The MCKP is related to the more conventional binary knapsack problem, sometimes denoted as 0-1KP, where each item in a set has associated benefit and weight values. The problem is to pack the knapsack such that a threshold weight is not exceeded, but the total benefit is as high as possible. In the MCKP variant, a set of sets is defined, where one object in each set must be chosen for inclusion. Again,

each object has associated benefit and weight values and the knapsack must be packed to maximize benefit subject to the total weight constraint. If all required objects fit, the solution is optimal. If not, the optimization problem must be solved. The solution to the MCKP is known to be NP-Complete [66]. Funkhouser and sequin use a greedy algorithm in [66], that packs items based on decreasing benefit/weight ratio, where benefit to the image is considered over the cost of rendering it. The greedy packing solution is at least half optimal [66]. Funkhouser and Sequin derive an incremental variation of this algorithm which exploits frame coherence in the solution, though its half optimality has been disputed by Mason and Blake [144], who derive a modified version of the algorithm by Funkhouser and Sequin [66] to support level of detail hierarchies. In their system, an object at one resolution is replaced by others at a higher resolution. It is claimed that their derivation is equivalent to a greedy solution to the MCKP which has a worst case complexity $O(n\log n)$ for n objects and is guaranteed to be at least half optimal.

Realtime controllers need to consider the objects in a scene when rendered from the current view point. They must estimate the importance of each object to the over all image, or more accurately, the benefit if each were rendered at different LODs. To do this, they must be based on a chosen philosophy of visual importance for an application, with corresponding metrics. This task is common to controllers which select pre-computed LODs or those which use selective refinement at runtime. Regardless of type, the algorithm must have low overheads to maintain interactive frame rates and make sure that the cost of using the LOD system as a whole doesn't outweigh the benefit by resulting in slower frame rates. It must also be born in mind that other processes will be taking place in an application and that the rendering system will only be a fraction of the tasks performed by the software as a whole.

Advanced visual metrics are out of scope here, but will be discussed briefly. There are a number of publications on perception and visual attention, some of which have developed models of relative importance to the human visual system. Many are too expensive to use in realtime, such as a system by Bolin and Meyer [19] based on image sampling, e.g. for ray tracing using a model of visual importance with wavelet image synthesis. Many visual models attempt to take *contrast* and *spatial frequency* into account based on achromatic methods, which are considered to be important factors in low level visual

attention. In particular, a technique by Luebke and co-workers in [139] uses a hierarchy of vertex groupings to provide multi-resolution models. At each change of LOD, the impact of the local surface change to the viewer is modelled in terms of contrast and spatial frequency. Changes can be traded off based on their visibility.

There are notions of visual importance common to many controller systems, whether they are selecting LODs for objects or whether to refine surface regions. These metrics are light weight when compared to advanced models of the visual system. They include:

1. Screen space size under perspective projection
2. Distance
3. Obliquity to viewer
4. Angular velocity
5. Peripheral degradation
6. Illumination
7. Semantic importance
8. Hysteresis

Let's first look at each of these and then look at how they can be put together to make a combined decision. The **screen space size** under a perspective projection is by far the most common and important metric used in controllers, particularly selective refinement systems. It is generally perceived that the larger an object is in an image, the more important it is to the structure of the image as a whole. Another way of looking at this is that the larger an object is in screen space, the more important it is to break it down to higher LODs. These subtly different views are applicable to static LOD control systems and selective refinement systems respectively. All controllers discussed here have a concept of screen space size.

It may seem that **distance** is a redundant term if screen space size is used. This isn't the case however, as proximity to the viewer can dictate the rate of parallax which is particularly important in coherence based systems or image caching schemes.

Obliquity, the angle of a surface to the viewer, is important in selective refinement schemes as it is a metric used to bias silhouette curves and interior curvature. Examples of selective refinement systems which use this method include [102], [126], [139] and [171].

When an object moves, it can become blurred to the viewer. It is widely accepted that detail may be reduced in this case. Therefore, the level of detail required is taken as inversely proportional to the **Angular velocity**. Object motion is considered in [66], [126], [136], [165] and [182].

Peripheral degradation refers to the possibility for reducing detail in the eye's peripheral region. The eye's fovea is the central region of the eye's image subtending approximately 2° from the line of view. Within this region, higher resolution and chromatic responses are experienced, in contrast to the peripheral region which is primarily achromatic. Small receptors called *cones* on the retina are responsible for chromatic response and are concentrated within the fovea. Achromatic response is provided by *rods* in the retina, which are rarer in the fovea but much more common in the periphery. An examination of whether peripheral level of detail reduction can be used in the context of a search task is given by Watson [208], where it was concluded that both image resolution and colour can be degraded by almost half in areas outside the fovea without negative effect. Peripheral degradation is commonly discussed in the literature, but rarely used in practice because it is difficult to map their behaviour to the display used. Monitors and LCD displays are by far the most common for graphics applications. Though the user may fixate in common ways, it is still possible to observe peripheral regions of the screen easily due to its small and variable field of view. Head mounted displays (HMDs) are also commonly used, but typically have a small field of view and the incorporation of eye tracking to assess user fixation is less common. Cave like environments offer a much wider field of view and may be potential candidates, but eye tracking may still be required. Peripheral degradation is commonly discussed in literature but strong evidence for its applicability is rare. Examples include [35], [126], [139] and [182].

Illumination conditions are occasionally considered by selective refinement systems as the specific lighting conditions are usually not known until runtime. Regions of the image containing substantial changes in illumination, particularly when combined with changes in surface colour may result in visible boundaries which might benefit from representation at higher resolution. Silhouette boundaries and specular highlights are a good example. Xia and Varshney state that a method catering for illumination conditions is used for polygonal simplification in [221] though the method is not described. A method

for refining regions of polygonal surfaces to render specular highlights in the absence of complete phong shading is also presented by Cho and Woo [31] which is not specific to LOD control.

Semantic Importance is an estimate of what features of the scene a viewer is focused on. There are likely to be low level and high level semantics which are applicable. Low level would include, for example, objects which exhibit creature motion and human faces as these are common tasks for the brain. Higher level semantics are likely to be very task oriented. If the purpose of an application is for analysis, the user's focus may be centred on whatever they are directly looking at. For a maintenance application, it may be a small set of objects involved in interaction. For non gaze based semantic importance to play an part in LOD control, it is necessary for type information to be included in the design of scenes in a way that is integrated with rendering. Semantics are considered in [66] and [182].

Lastly, **hysteresis** refers to the requirement for smooth transitions between levels of detail without noticeable switches, referred to as the *popping* effect. In LOD, if updates are performed quickly between similar LODs, the situation is acceptable. If LODs are not switched frequently, large changes may occur.

If separate lightweight metrics are formulated, it is necessary to combine them to make decisions. Luckily, there are no strong conflicts to resolve between them. However, it is necessary to combine their influence in some way and make global trade-offs between objects or surface regions for resources.

One of few methods in the literature has been given by Funkhouser and Sequin [66]. To provide an over all benefit value, they take a product of all terms:

$$Benefit(O, L, R) = Size(O) \cdot Accuracy(O, L, R) \cdot Imp(O) \cdot Focus(O) \cdot Motion(O) \cdot Hysteresis(O)$$

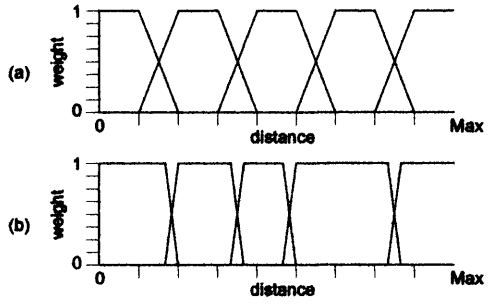
(EQ 2)

where O is an object, L is a level of detail and R is a specific rendering method. All metric domains are defined over the $[0...1]$ interval and have equal weight. The $Size$, $Motion$ and $Hysteresis$ terms are as described above. The importance Imp is a semantic

weighting and the *Focus* is the area looked at by the viewer. The *Accuracy* is dependent on the rendering method used, which in their system can be flat or Gouraud shaded. Though this method if combining terms is simple, it is fast to evaluate. Additional functions could be added to each term if required. This benefit function is useful for static LOD control, but its use for selective refinement systems may prove quite expensive. A very similar product combination function is given by Lau and Green in [126] who unusually include a term catering for depth of field effects.

A more complex system based on fuzzy control theory has been given by Schraft and coworkers [182]. Each of their metrics are split into several sets, for example, applying linguistic meaning to object distance sets may give {*very near, near, medium, far, very far*}. An example of fuzzy set membership for object distance is shown in Figure 15(a). This is a hypothetical configuration for purposes of explanation. These sets are defined over an interval deemed appropriate for the metric type. Set membership functions are defined using trapezoids which overlap and span the interval. These provide a weighted multiple set membership. Weightings are defined between [0...1].

FIGURE 15. Hypothetical fuzzy set membership functions before (a) and after (b) optimization



A fixed set of production rules are used to combine metrics, e.g.:

if (object is near AND object is large) then Increase LOD

if (object complexity is low AND object is small) then Decrease LOD

Fuzzy set membership functions are then optimized using a genetic algorithm. Figure 15(b) shows an example. A common method of crossing and mutation is used between generations of the populations. The fitness function used is not described in detail in [182], but is based on an analysis of frame rate consistency. This system is particularly

interesting as it is a global approach which does not make explicit trade-offs between objects. It attempts to establish performance and consistency as a resulting behaviour of local choices. The authors claim that the optimization can be performed quickly enough for realtime adaptation. Though mean frame rates appear to be fairly stable, deviations appear to be quite large.

Various viewpoint dependent selective refinement control systems have been developed in recent years. These are mostly based on hierarchical vertex grouping, generally for polygonal rendering. These include [102], [126], [136], [139], [181], [221] and [224].

An image based control technique is given by Cohen, Olano and Manocha [38] that reduces surface geometry appearance to texture and normal maps, applied to lower LOD models, obtained using edge collapses. Deviation between textured appearance approximation and original surface points is evaluated using texture deviation metrics that permit error bounds in image space. Lindstrom and Turk also use an image based approach [134] [132], evaluating the error of an edge collapse by comparing images of the full LOD model with the approximating model.

Virtually all point based rendering systems use viewpoint dependent refinement controllers, where the most common error metric for image quality is the image space size or density of point samples under perspective projection. Clark [35] was the first to explore this form of hierarchical selective refinement method for arbitrary geometric representations. An early point based viewpoint dependent image sampling system was developed by Levoy and Whitted [130]. Later, the QSplat system by Rusinkiewicz and Levoy's [171], [172] has become a popular reference point in the field. Later examples include [22] [158] [72] [83] [127] [169] [194] (see Section 2.9.4). and in a GPU cached form in [48].

A small number of systems attempt to combine level of detail systems with occlusion culling. The reader is referred to Section 2.7 where these systems are discussed.

2.4 Visibility Ordering

2.4.1 The Problem

Visibility ordering is the control of the order in which primitives are processed, either within the rendering algorithm before rasterization, or at rasterization time. The standard requirement for visibility ordering is to establish a depth ordering from back to front in camera space for the rasterization of primitives. This ordering is correct for the occlusion of opaque primitives and is a correct accumulated compositing order for scenes with translucent or transparent surfaces. In some cases, a front to back compositing order is used with transparent surfaces, such that rasterization for a pixel can stop when the opacity of a pixel reaches a maximum [163].

Algorithms may also require depth ordering for purposes such as occlusion culling, where a front to back ordering is sometimes required. The Canopy algorithm requires a front to back traversal ordering of the scene in camera space for occlusion culling. It can also benefit from a back to front ordering for approximate depth ordering for primitive rasterization.

This section will briefly discuss some techniques used in the literature for the depth ordering of primitives for the purposes outlined above.

2.4.2 Visibility Ordering Techniques

Early wire-frame rendering side stepped the depth ordering requirement for rasterization occlusion using hidden surface removal algorithms to eliminate lines that were not visible [63]. Depth complexity in these systems was low, as hardware was only powerful enough to render a few objects at one time. As such, objects could be sorted for depth. As technology progressed, scenes became more complex and in the mid 1980s, even home systems became capable of rendering solid 3D polygons. Hidden surface removal using *back face culling* then became a popular method along with object depth sorting, a method used in many games. This asserted a constraint that objects were broken into convex polyhedra to function correctly.

The technique of rendering components from back to front is generally termed a *painter's* algorithm as this is sometimes the depth order used in artwork. There is a specific method known as the painter's algorithm that sorts polygons for depth [63].

By far the most popular solution for obscuring for rasterization has been the z-buffer, introduced by Catmull [25] which is a general purpose solution for use with most primitive types and has become a standard feature in polygon rendering pipelines. High end systems from SGI first introduced hardware implementations, with games level systems incorporating them in the mid 1990s. The z-buffer is a memory buffer with one depth entry for each pixel in the frame buffer, which represents the distance of the closest surface that has been rendered to the pixel. Initially, all depth values in the z-buffer are set to a very large or invalid value. As primitives are rasterized to the display, their depth under a perspective projection is compared to the pixel's depth in the z-buffer. If further away, no writes are made to the z-buffer or image buffer. If nearer, the test succeeds and a write is made to both the image and z-buffer with corresponding values.

The z-buffer method is only really accurate using 32 bits per pixel integers or floating point values to offer accurate depth resolution. The z-buffer can also suffer from zig-zag effects known as *flammering* if two surfaces quantize to the same depth, with different primitives being chosen for display at different pixels or at the same pixels between frames. Primitive index based priority methods can overcome this problem.

The z-buffer is a pixel level occlusion culling system with $O(n)$ complexity, where n is the number of primitives. Moreover, it is a $O(a)$ complexity system, where a is the total surface area of all n primitives, because each must undergo full rasterization. The need for the z-buffer in standard rendering highlights the inefficiency that results from large levels of *over-draw* in scenes with high depth complexity. Techniques such as occlusion culling can substantially reduce this over-draw.

An alternative method of note called *binary space partition* (BSP) trees was given by Fuchs, Kadem and Naylor [64], later applied in the games field in the 1980s[1]. This method stores the scene as a binary tree, where each node is associated with a plane

1. The first consumer product known to the author to use BSP-Trees was the game Carrier Command by Realtime Games written by Ian Oliver and Graeme Baird, published by Rainbird in 1988.

which partitions the scene into two convex regions. Unbounded areas at the scene periphery are also considered to be convex. Scene primitives which lie on or close to this plane may be stored in the node, or alternatively, primitives can be pushed down to the leaves of the binary tree. Each of the branching node's two child trees represents a sub-tree of two primitive sets and the method recurses. Any primitives which intersect a plane are split into two halves on either side of the partition. The leaf node partitions contain all primitives which are left. The result is a tree which partitions convex cells. A simple application of a vertex to the partition's plane equation can state whether the point is in front or behind the plane. If in the plane, it can default to one side or the other.

As such, for a given viewpoint and BSP-Tree node it is possible to tell which child tree is closer and which is further, as the side containing the viewpoint is always nearer. This allows simple $O(n)$ time determination of a back to front visibility ordering for conventional rendering, or front to back to support occlusion culling algorithms, where n is the number of polygons. With regard to rendering algorithm properties discussed in Section 1.1, the technique has gained rendering speed and traded off dynamic potential and memory to achieve this. BSP Trees are known to have scalability issues for construction. The upper bound for auto partitioning is $O(n^2)$ for both spatial and running time complexity, where a set of n polygons are split over all partitions.

Octrees [109] can be interpreted as a BSP tree [64], each cell having three partition planes, forming a K-D style tree [15]. Their ordering can therefore be inferred by the viewpoint's position in relation to these planes in the octree in the same way as the BSP tree. A regular 3D grid also has an ordering defined by all separating planes in the grid space and depth ordering can be found quickly.

Many point based rendering systems, particularly those based on the *Surfels* system by Pfister and colleagues [158] use *visibility splatting*, which is based on a multi-pass z-buffer technique (see Section 2.9.5.5) resulting in linear scalability.

Laur and Hanrahan use an octree derived depth ordering their hierarchical splatting system [127], as to Coconu and Hege in their point based system [36] to incorporate transparency, rather than use the more common visibility splatting method. Octree ordering is

used by Green, Kass and Miller's *hierarchical z-buffer* technique [78] for occlusion culling based on an image space z-pyramid (see Section 2.7).

More generalized BSP tree structures are used by Greene [77] in polygon tiling system for occlusion culling, combined with an octree of BSP trees.

A 3D grid system is used by Hillesland and colleagues [97] for front to back traversal in a straight forward occlusion culling system for polygon models, extended to a hierarchical spatial subdivision system. Stamminger and Dretakis [194] use a front to back ordering for grids using their $\sqrt{5}$ sampling scheme for surface refinement and define an occlusion culling algorithm based on this for terrains.

One of the few systems to integrate level of detail and occlusion culling by El-Sana, Sokolovsky and Silva [54] uses a 3D grid fitted to the bounding volume of the scene with a front to back traversal for occlusion culling.

Visibility ordering for arbitrary space meshes has been addressed by Williams [215] and Silva, Mitchell and Williams [215] using Directed Acyclic Graphs (DAGs) of local cell adjacency. The basic concept of directed adjacency graphs has similarities with the *image graph* data structure that is central to the algorithm described in this thesis.

2.5 View Volume Culling

2.5.1 The Problem

A detailed scene, with a viewpoint positioned well within the scene, will have many regions that are not visible, simply due to the camera's position, orientation and limited field of view, as defined by the view volume (see Section 1.2). Even a locale may have large regions that are not being viewed in the image.

View volume culling is a form of visibility culling that quickly culls objects or primitives that are outside the view volume and therefore not visible.

2.5.2 View Volume Culling Techniques

Many techniques have incorporated view volume culling, so this section will only address a small number of commonly used methods.

Frequently, whole objects are view volume culled by testing their bounding volume for containment or intersection with the view volume. For n objects this is a $O(n)$ solution. Addressing all objects in the scene is not efficient. Therefore, many systems use object space bounding volume hierarchies to approach a $O(Logn)$ time method. The first known proponent of this method was Clark [35], who was the first to create a hierarchical data structure which can be culled to the view volume in an expected Log time.

Probably the most efficient systems for view volume culling are those involving volume spatial representations such as voxel grids or forms of space meshing such as surface conforming Delaunay [42]. These systems can propagate through the region in the view volume, where cells on the view volume pass spatial coherence information to their neighbours for testing. The majority of cells inside the view volume are never tested and those outside are not traversed. As the number of cells becomes denser, the ratio of tested cells to interior cells tends towards zero.

Hierarchical bounding volumes have been used widely, mostly based on existing object hierarchies. Bounding spheres have been used for culling by Clark [35] and Rusinkie-

wicz and Levoy [171] [172]. Other forms include axis aligned bounding boxes (AAB) [75] and oriented bounding boxes (OBB) [75].

An efficient probabilistic method for a set of objects has been given by Slater [191]. This method partitions objects into sets which are outside, inside and intersecting the view volume. Objects outside the volume are partitioned into sets based on their distance from the view volume and are then sampled accordingly. This method has shown to be approximately twice as fast as bounding box methods for scenes tested.

Point based systems, using their respective hierarchical data structures, often incorporate hierarchical view volume culling using bounding volumes. Examples include Rusinkiewicz and Levoy [171] [172], Pfister and colleagues [158], Ren and colleagues [168], Wand and Straßer [206], Pajarola [151], Xu and Chen [222], Botsch and colleagues [22], Coconu and Hege [36], Hopf, Luttenberger and Ertl [98], Chen and Nguyen [29], Duguet and Dretakis [52] and Gobbetti and Marton [72]. Kruger and colleagues [124] use multiple resolutions of a hexagonal close packing (HCP) grid for hierarchical culling. Notably, the *sequential point trees* system by Dachsbacher and colleagues [48] sacrifices hierarchical frustum culling techniques for a flattened data structure suitable for GPUs.

2.6 Back Face Culling

2.6.1 The Problem

Polyhedral objects in the view volume only show approximately half of their surfaces to the camera at any one time, with angles between the surface normal and view vector from primitive to the camera's center of projection over the interval $[0...\pi/2]$. Angles in $[\pi/2...\pi]$ would show their backs to the camera and can therefore be culled to reduce the number of processed primitives by about a half.

2.6.2 Back Face Culling Techniques

Standard back-face culling for polygon models [63] has been used widely and was initially even used as a process for hidden surface removal, with depth sorted convex objects for correct occlusion of front facing primitives. This was an especially common technique in games in the 1980s. A back face test succeeds iff:

$$V \cdot N \leq 0 \qquad \text{(EQ 3)}$$

where V is the view vector from the primitive to the camera's center of projection and N is the surface normal vector. If this test succeeds, the primitive is back facing and culled, otherwise it is front facing and is rendered.

Back face culling is present in hardware pipelines. Given a set of n primitives, the system has $O(n)$ complexity if all n are tested. To improve this complexity, hierarchical methods have appeared in the literature and are also now quite widely used. Kumar and colleagues [125] define a cone of normals that conservatively bounds a set of normals associated with a set of child primitives. Hierarchical tests typically then measure if a normal cone is wholly *front facing*, *back facing* or mixed. If wholly *front* or *back* facing, no children in the hierarchy need to be tested. If mixed, the child nodes do require further testing as they lie on a silhouette region.

Given a set of polygon or point primitives with normals, a set of half spaces is defined, with a front half space on the normal side and a back half space on the opposite side of the primitive. The region of space defined by the intersection of all front half spaces defines an invariant region where all primitives are front facing, if the camera's center of

projection is contained within it. Similarly, the intersection of all back half spaces defines an invariant region where all primitives are back facing. The complement space outside the union of the front and back regions defines a *mixed* front and back region. A bounding scheme including cones for these regions in hierarchies are given by Kumar and colleagues [125], also used by Fleishman [61]. Grossman and Dalley [80] and Zwicker [229] use a *visibility cone*, with axis equal to the mean normal vector, that intersects the mean point in a point set. A normal cone apex is then chosen such that it bounds both the normals and the point samples. This technique is only defined for a normal vectors over the unit hemisphere. This technique is used again by Pfister and colleagues in the Surfels system [158] and Guennebaud and Paulin [84].

Hierarchical back face culling allows amortized tests to be carried out for a subset of the scene. If found to be back facing, no children in the hierarchy need be traversed. Hierarchical normal schemes including normal cones have also been used by Shirman and Abi [186] for Bezier surfaces and Zhang and Hoff [226] also define a fast normal cluster based system for fast per primitive tests, also used by Luebke [138]. Luebke and Erikson [136] and Sander and colleagues [173] use normal cones to identify silhouette regions.

Point based rendering systems using hierarchical back face culling include QSplat by Rusinkiewicz and Levoy [171] [172], POP by Chen and Nguyen [29], Confetti by Pajarola [151], Botsch and colleagues [22], the Layered Point Clouds of Gobbetti and Marton [72], Krivanek [123] and in the Sequential Point Tree system by Dachsbacher and colleagues [48].

The normal cone suffers substantial efficiency problems. The cone is not an efficient bounding shape, because a single radical normal affects the whole bound, not just in the direction of the normal's deviation from the cone axis, but in all directions about the axis due to the cone's increased angle. For example, normals for an un-capped cylinder will have the same cone as a sphere. Alternatives such as a frustum would may be more suitable, with potentially tighter bounds.

A simple normal cone system, with apex centered at the position of a bounding volume that anchors all normals in the normal cone is not sufficient to conservatively test for potential visibility by simply testing the centroid axial normal vector with the view vec-

tor. This is because the approximated child primitives may take any position within the volume, affecting the view vector to the primitive and hence the back face culling calculation. We have independently developed a conservative solution to this problem, equivalent to the *spatialized normal cone* by Johnson and Cohen [111] that takes account of the maximal angles that could be present in the viewing conditions, given a bounding sphere, normal cone and view cone (see Section 5.6.2). This system conservatively defines whether any point in a bounding sphere with a specified normal cone is wholly front facing, back facing or mixed facing. This technique allows hierarchies to be constructed without direct access to any leaf levels of the hierarchy for scalability and speed of reconstruction for dynamics.

Johnson and Cohen [111] demonstrate speedups of up to about 30x using spatialized normal cones in a hierarchical system for silhouette extraction. The authors also show how the spatialized normal cone technique can be extended to fast point and model nearest distance tests and area light source umbra and penumbra boundary detection.

2.7 Occlusion Culling

2.7.1 The Problem

Ideally, only the scene detail that will be present in the image should be addressed by the rendering algorithm. There may be a large number of depth layers present in the view volume. Standard rendering pipelines simply render all depth layers. Hidden surface removal is usually achieved using a z-buffer. Moreover, most systems also shade each fragment that passes the depth test for visibility at that stage of rendering. Back face culling halves the number of depth layers for polyhedra, but the depth complexity of some scenes can still be large. This method therefore has $O(d)$ complexity, without LOD aggregating layers, where d is the number of depth layers in the view volume, due to pixel over-draw. Due to this dependency, if d were conceptually infinite, the algorithm would never halt.

An occlusion culling system attempts to achieve sub-linear complexity by efficiently identifying objects or surface regions hidden behind others and culling them as early as possible, with as high a degree of accuracy as possible. Ideally, culling will only identify the first depth layer and discard all further depth layers to approach zero over-draw. This ability can be extended to point light source shadows (see Chapter 4).

To remove the dependency on the number of depth layers in the view volume d, it is also necessary to detect when the image is complete, such that no other scene regions in the view volume are traversed once this condition has arisen, but this is not often achieved in existing systems.

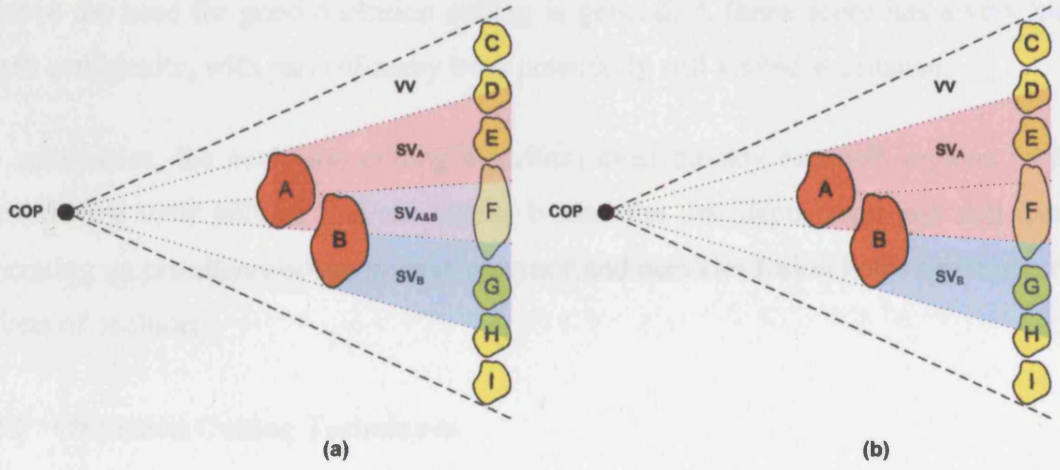
Two additional benefits of accurate occlusion culled solutions are that over-draw is minimized, such that shading is calculated as a second pass in a manner more similar to deferred shading [81] and secondly, that the visible surface set can be cached for reuse of some types of data in future frames.

Here, we will define three states of occlusion, *visible, partially visible* and *occluded.* An occlusion scenario is shown in plan view in Figure 16(a). The view volume VV contains a two occluders A and B that cast occlusion volumes SV_A and SV_B respectively, due to

the perspective projection of A and B from the center of projection COP. Due to the similarity with a light source at the COP casting shadows, these volumes are also equivalent to shadow volumes. The intersection of these volumes is $SV_{A\&B}$. A range of far objects are also contained within VV, as objects $C...I$. Object C is not in the volumes of either A or B and is therefore considered *visible*. Object D is *partially visible* because it is only partially occluded in SV_A. E is *occluded* by A in SV_A. Similarly, object G is *occluded* by B in SV_B, H is *partially visible* in SV_B and I is *visible*.

Object F presents a particular problem. With respect to occluder A, it is *partially visible* due to partial occlusion in SV_A. Likewise, with respect to B, F is partially visible in SV_B. However, on aggregate, F is *occluded* by the union of occluders A and B in a union volume $SV_A \cup SV_B$. The detection of cases C, D, E, G, H, I are more straight forward and are detected by most occlusion culling algorithms. Case F however, is less easy due to the need to cater for the effects of multiple combined, or *fused* occluders.

FIGURE 16. View volume VV with occluders A and B with potential occludees C to I



Occlusion culling systems can either operate at object level or at primitive level. At object level, whole objects are compared and potentially culled. Primitive level culling is potentially more versatile because different regions of an object can be visible or culled.

An object level system must render the entirety of objects classed as *visible* or *partially visible*. In Figure 16, objects C, D, H and I are not culled. Objects E, F and G classed as

occluded should be culled. A primitive level system may treat D and H differently, where only *visible* or *partially visible* primitives are rendered, the rest being culled. Object F may only be classified as *occluded* and culled in systems using occluder fusion.

It should also be noted in Figure 16(a) that occluder A partially occludes B and therefore B is itself only *partially visible* due to SV_A. Therefore, in a system that operates on primitives, occluded regions of B in SV_A may be culled and will not cast the intersection volume $SV_{A\&B}$, as shown in Figure 16(b). Only a reduced volume SV_B is cast.

Depth complexity in scenes can vary substantially. A room may have only a few depth layers, for example, 3 or 4. A large building with connected rooms however, may have a much higher depth complexity. Scenes with a low depth complexity may be rendered more slowly when occlusion culling is used, due to the overhead of managing potential occlusion that is not present.

The depth complexity may be reduced by the locale management system, if for example, a cells and portals technique is employed (see Section 2.2)., but this is unlikely to remove the need for good occlusion culling in general. A forest scene has a very high depth complexity, with parts of many trees potentially still visible at distance.

To summarize, the occlusion culling algorithm must quickly establish regions of the scene in the view volume that are hidden behind the first depth layer and cull them. Operating on primitives brings greater accuracy and occluder fusion helps aggregate the effects of occluders.

2.7.2 Occlusion Culling Techniques

Various forms of occlusion culling technique have been developed. This section will examine some of the more prominent algorithms. For further information, a survey and taxonomy is given by Cohen-Or and colleagues [40].

Occlusion culling algorithms vary in the features they offer and the types of scene they work with. It is costly to calculate an exact set of objects or primitives in a scene that are visible or partially visible. Therefore, most methods calculate a *potentially visible set*

(PVS). The PVS may be *conservative* in that it contains at least the set of objects or primitives that have visibility. Alternatively, *approximate* algorithms are not conservative and may omit visible objects or primitives to achieve greater speed or simplicity. Examples include Klosowski and Silva's PLP [121] and Andujar et al.'s HVS [9].

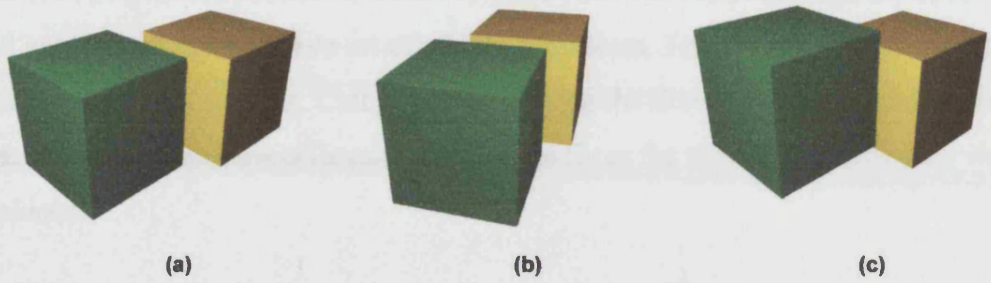
It is fairly common for occlusion culling techniques to be less reliable than conventional rendering for scenes with low depth complexity. This is because calculations are performed which do not result in a reduction in the calculations required to render the image. However, as the depth complexity increases, there is a cross over point at which time the occlusion culling algorithm becomes more efficient than a standard brute force renderer.

Occlusion culling algorithms are generally world/object space, or image space based. Some use purely geometrical calculations to determine visibility, whereas others perform occlusion culling as the image is constructed. Algorithms often use either a full set of occluders, or a chosen subset.

Two common types of algorithm are based on *from-point* and *from-region* visibility [40]. From-point algorithms calculate a PVS for a single viewpoint. This may be view direction dependent or independent. From-region algorithms calculate regions of space in which a single PVS solution is valid and invariant, either in a 2D, 2.5D or 3D domain. From-region algorithms often rely more heavily on pre-processing to provide stored results for speed and sacrifice surface dynamic potential (see Section 1.1) in scenes with movement. Because from-region methods are pre-calculated, they can be aware of the visibility qualities of neighbouring regions which aids pre-fetching of data, e.g. for storage or network streaming. From-point methods rely more heavily on runtime processing and may not be as fast, but are in some cases applicable to dynamic scenes.

A technique known as the *aspect graph* developed by Koenderink and Van Doorn [122] is important as it defines all occlusion solutions possible for a given polygon based scene. Given a viewpoint and a scene, the viewspace in which the camera moves is partitioned into cells. Within each cell, all possible positions and view directions have the same set of visible primitives. These primitives have the same overlapping connectivity in screen space, described by an *image structure graph* (ISG).

FIGURE 17. Different aspects of the same scene. (a) & (b) have same aspect. (c) is different.



Example aspects of a scene with two cubes is shown in Figure 17 for three views. The first two (a) and (b) have the same aspect as only the top and side faces of the front cube overlap the nearest face on the far cube in image space. The third view (c) has a different aspect as the same two faces of the near cube are now overlapping the top face of the rear cube. Note that in this example, the set of visible polygons is the same in all aspects. The viewspace is partitioned into cells, termed the visible space partition (VSP), where each cell is an ISG invariant region because its ISGs are isomorphic. Each partition between cells represents a *visual event* where the ISG has changed. The aspect graph is the graph of the connectivity between cells. The aspect graph system is however, not very scalable. As the number of primitives in the scene grows, the number of cells increases dramatically. The worst case complexity is $O(n^9)$ in three dimensions using a perspective projection [40], so the method is not practical for use in large scenes.

A technique which is similar to a relaxed form of the aspect graph has been given by Cohen-Or and colleagues in [41]. This method pre-computes a viewspace partitioning into cells. Each cell is associated with a conservative super set of visible objects, rather than primitives. Their method is based on the concept of a strong occluder which is a single object that wholly occludes another. An object is occluded if there is a strong occluder which occludes it from all positions within the a cell. It is claimed that many environments have numerous objects which are strongly occluded and that the use of an approximate super set of visible objects is not much less efficient than using exact visibility determination.

Another method is given by Coorg and Teller [43] [44] which is also related to a simplified form of the aspect graph, where the viewpoint space is divided into cells. Far fewer

cells are used in comparison to an aspect graph, as the partitioning is based on a subset of large convex polyhedral occluders, which occlude other convex polyhedral surfaces. The method generates a conservative set of visible primitives. The set of cells for the whole scene are generated at runtime. Cell planes containing the first viewpoint are initially calculated, with planes for new cells calculated on the fly as the viewpoint crosses the visual event planes.

Dynamic octree spatial partition techniques are addressed by Sudarsky and Gotsman [196] who define temporal bounding volumes for visibility between objects in a form of from-region occlusion culling, achieving a speedup of nearly 3x for a small scene with 5,745 polygons and a single small dynamic object.

Interior scenes have very specific structure and occlusion properties. Rooms are connected together in a known way. Walls are mostly opaque and visibility is only possible through what are usually simple shapes such as doorways and windows. This context is an example of a *cell* and *portal* structure, related to locale management (see Section 2.2). Rooms are usually referred to as *cells* and the doors and windows are portals between cells. Algorithms which are specifically designed to use this scene structure have been developed, which are also methods based on spatial analysis. Teller and Sequin [201] developed a from-region system for pre-processing axis aligned architectural walkthrough scenes. The model is partitioned based on the major walls of the scene, using a K-D Tree, resulting in convex cells. An adjacency graph between cells is used as a basis for cell to cell visibility calculations through portals. At runtime, the set of potentially visible cells are also culled based on the view volume. If a cell is determined to be visible, its geometry is rendered.

A similar from-point technique based on cells has been given by Luebke and Georges [137]. In each frame, the cell containing the viewpoint is rendered. The system then identifies all portals in the view volume and creates new sub-view volumes based on the axis aligned bounding box of each portal, clipped to the parent sub view-volume. The cell visible through each portal is rendered, clipped to the sub-view volume. This process is carried out recursively.

Hudson and colleagues [108] use a relatively simple from-point shadow volume algorithm, selecting large occluders at runtime, using the standard hierarchical bounding volumes of the provided scene. A set of occluders must be selected and a frustum fitted to each occluder. The hierarchy of bounding boxes are then tested against the shadow volumes and culled. The authors report that approximately half of the scene's primitives in the view volume are culled conservatively.

The conservative from-point algorithm by Muller and Fellner [148] constructs a binary tree hierarchy of axis aligned bounding boxes of a given set of objects at the leaves, with occluder selection based on ray casting in a quad tree subdivision scheme in image space, with frame coherence speedups. Bounding volumes are sorted for depth and traversed in front to back order. Hardware based rasterization is used to establish occlusion at pixel level. Tracing from visible occluders to the root then quickly establishes visibility of a subset of nodes in the hierarchy. The hierarchy is then refined from the root downward, rendering leaf nodes if encountered, whilst testing and potentially culling bounding volumes in the hierarchy. Occlusion tests for nodes during hierarchy refinement are also disabled temporarily to exploit coherence. For a city model with about 850,000 triangles, an order of magnitude speed-up is achieved for a highly occluded walkthrough of the streets.

The Shadow Volume BSP (SVBSP) tree by Chin and Feiner [30] uses a standard BSP tree to traverse in front to back order from a light source (or viewpoint). Each polygon resulting from this ordering is inserted into an SVBSP tree that describes the occlusion volumes formed by planes defined by a light source (or viewpoint) and the edges of each polygon. Those in the volume may be discarded, while those visible are inserted into the SVBSP tree. This can be used as both a shadow determination or occlusion culling algorithm with $O(nLogn)$ complexity. Naylor [149] uses a 2D BSP tree to describe an image, projected from a 3D BSP tree. Dynamics for SVBSP trees were considered by Chrysanthou and Slater [32].

Bittner, Havran and Slavik [17] define a from-point algorithm that selects a set of convex polygons as occluders and construct an *occlusion tree*, merging occlusion volumes of separate occluders, based on the shadow volume BSP tree by Chin and Feiner [30]. A spatial hierarchy allows hierarchical culling. If nodes are fully visible or fully culled, this

state applies to the sub-tree of the hierarchy. Bittner and Havran [16] define a modification of [17] to reduce occlusion tests, exploiting frame coherence.

Image space algorithms include the *hierarchical z-buffer* by Green and co-workers [78] and the Hierarchical Occlusion Masks of Zhang [228]. The Hierarchical Z-Buffer method by Greene, Kass and Miller [78] has become a classic from-point occlusion culling algorithm due to its simplicity and feature set. However, it is generally accepted that the method is not effective without explicit hardware support. The authors claim an order of magnitude speed-up with scenes they have tested. It is a good example because it exploits the three major forms of coherence often sought. Their algorithm takes advantage of object space, image space and temporal coherence. Object and image space coherence are hierarchical and allow early culling decisions to be made at lower expense, a concept also intrinsic to Zhang's Hierarchical Occlusion Maps [228] which will be discussed shortly. Green's method constructs an octree of an object's polygons, starting from a bounding cube which contains the original model. If a small number of polygons are contained within an octant, the octant becomes a leaf node, otherwise, polygons that are wholly or partially contained by child octants are passed down to these new child octants. A second pyramidal z-buffer hierarchy is constructed in screen space. Starting with a conventional z-buffer at the bottom of the pyramid, groupings of four pixels are made to form the next lower resolution. This process continues until the root of the pyramid is reached, forming a single pixel. Each pixel represents the furthest z-value that its screen space area covers in the original z-buffer. Each lower resolution forms a summarization of the essential depth information in its higher resolution. To see if an octree octant is visible, polygons used to describe the octant's faces are tested against the z-pyramid at an appropriate resolution. An octant is culled if all its face polygons are occluded. If the octant is not occluded, the algorithm recurses to the child octants in front to back order. If a leaf node is reached, its contents are rasterized into the z-pyramid, which is an expensive operation.

The Hierarchical Z-Buffer technique also makes use of frame coherence. Between frames, some octants will remain visible, others which were visible will become occluded, and others which were occluded may become visible. After the first frame, each frame inherits a *temporal coherence list* of visible octants that were rendered in the

last frame and are again rendered in the current frame. The z-pyramid is then constructed using the resulting z-buffer. The main algorithm is then used to verify and discover octant visibility, requiring less recursion. Rendered octants are also then checked for visibility to remove occluded octants from the list. The primary difficulties with the algorithm are the requirements for expensive z-pyramid updates and memory reads during the rendering process, that are not well suited to current GPU architectures.

A second popular from-point image based algorithm is the Hierarchical Occlusion Map (HOM) by Zhang and co-workers [228]. Similarly, this technique uses spatial coherence in the form of an object hierarchy and screen space coherence using an image pyramid (HOM), similar to a mip-map, that requires no potentially expensive updates. As a pre-process, candidate occluders are chosen as a subset of objects in the scene based on size. At runtime, a subset of these occluders is chosen, based on those in the view volume. In each frame, occluders are rendered into a monochrome image at full resolution and intensity. An image pyramid is then constructed, using an averaging process to evaluate mean opacity up the pyramid. OpenGL hardware mip-mapping can be used. A depth buffer is also rendered. Given a bounding volume, a place in the HOM is chosen such that the bounding rectangle of the projection of the volume is equal to that of a HOM pixel. The rasterization pixels of the bounding volume are tested for opacity with a thresholding scheme at the HOM resolution. If the opacity is above an opaque threshold, a depth test then establishes if the occluder is in front of the occludee. If it is, the object is rendered, else it is culled. If the opacity of an HOM pixel is lower than the threshold, its child pixels in the next higher resolution are tested in a recursive, quadtree style process, with depth tests. If sufficient occlusion is not established, the object is rendered. This process terminates quickly and conservatively to attempt to keep the algorithm's running time low. In use, the HOM method is shown to provide higher frame rates in scenes of about 1M polygons which at times are faster by nearly an order of magnitude. Original bounding volume hierarchies are used, without pre-processing to derive more efficient hierarchies. The percentage of the scene which is culled can also be close to 90% in large scenes. The HOM offers approximate culling, ignoring holes, with occluder fusion. It is also suitable for two pass rendering for modern GPU architectures. The method does however, rely on occluder subset selection.

Hey and colleagues [96] also use an image based technique, using a flat occlusion grid at lower resolution than the z-buffer and carry out 'lazy' updates of the grid, only when queries are made, reducing the number of occlusion queries, when rendering hierarchical bounding volumes in an approximate front to back order. Speedups of up to an order of magnitude are shown against rendering without occlusion culling.

A straightforward image based method is given by Hillesland and colleagues [97]. An octree structure's cells reference intersecting geometry. A front to back rendering order renders the contents of cells. The hardware *GL_NV_Occlusion_Query* extension tests cells against already rendered geometry. Visible cell contents are rendered. Rendering time for a power plant model with 13M polygons is shown to reduce from 0.8s to 0.1s.

The from-point *conservative probabilistic layered projection* (CPLP) technique by Klosowski and Silver [120] is a less conventional, probabilistic algorithm with occluder fusion in object space. It is a conservative alteration of the approximate, non-conservative, time budget based PLP algorithm [121]. An octree or Delaunay mesh partitions the scene into cells. A visibility probability is assigned to each cell based on solidity, viewpoint position, direction and neighbouring cell geometry. A priority queue system incrementally renders the scene, based on occlusion probability in an approximate front to back order. The PLP algorithm has a first pass to establish an approximate solution, potentially with missing visible objects. This result is used as an occlusion mask in a second pass, where OpenGL hardware based visibility queries are used to establish the visibility of remaining missing scene regions for conservative rendering. CPLP is combined with out-of-core rendering in the iWalk system by Correa, Klosowski and Silva [45].

A smaller number of systems integrate level of detail and occlusion culling. The Berkeley Walkthrough system [65] combines cells and portals visibility (see Section 2.2) with discrete LOD models.

The MMR system by Aliaga and colleagues [7] combines several optimization schemes. A cell based locale management system allows for pre-fetch and *textured depth meshes* to approximate far geometry. Near geometry uses hierarchical LOD rendering and occlusion culling using selected occluders for each cell in a system based on the *hierarchical occlusion map* by Zhang and colleagues [228]. The system is only suitable

for static scenes. Tests show the system reducing a 15M polygon scene to about 200k polygons for rendering, achieving about 20 fps on a system with 4 195MHz CPUs.

The Gigawalk system developed by Baxter and colleagues [14] integrates the GAPS hierarchical level of detail control system of Erikson and Manocha [56] (see Section 2.3.3) with the *hierarchical z-buffer* [78]. A partitioning and clustering system is used to create a scene graph used for both level of detail and occlusion culling. Using a system with three 300MHz CPUs and 2 Infinite Reality 2 pipelines authors demonstrate frame rates of up to 50fps for a highly complex scene with 82M triangles, shown to be reduced to under 100k for rasterization. The technique is not suitable for dynamic scenes.

Andújar and colleagues [9] use an approach based on *hardly visible sets* (HVS). Potentially visible set (PVS) objects are arranged into several HVS's, with similar degrees of occlusion. Level of detail is affected by the degree of visibility of an object in image space. A set of occluders is approximated by boxes for occlusion estimation by the whole box, without occluder fusion. Partially visible objects are identified by the enlarging the occluder boxes, calculating overlap and removing wholly occluded objects from the resulting set. This process is repeated to discover sets with varying degrees of visibility.

The 3D grid based system by El-Sana, Sokolovsky and Silva [54] also combines visibility determination in a viewpoint dependent selective refinement LOD controller to prohibit high resolutions in occluded regions, where the heuristic to reduce LOD based on level of occlusion is also used. The static scene's bounding box is subdivided into cells and a solidity value calculated for each by either projecting the cell's geometry onto its faces or using ray casting to sample occlusion ratios through the cell. A front to back rendering order is then used.

Of particular note, due to its similarity with the Canopy system in this thesis, is the combined LOD and from-point, conservative occlusion culling system by Yoon, Salomon and Manocha [224]. Like Gigawalk [14], the GAPS HLOD system [56] is used, with frame coherent occlusion culling. GAPS is used to create a vertex hierarchy, then decomposed to a cluster hierarchy, used for view volume and occlusion culling. LOD control is based on edge collapse operators with a Hausdorff metric and image space pixel deviation. The previous frame's visible clusters are refined and used as the current

frame's occluders, for frame coherence. Cluster oriented bounding boxes (OBBS) are then tested for visibility using hardware occlusion queries and included if they have become visible. Performance on an Intel 2.8GHz Pentium 4 system with nVidia GeForce Ti4600 is shown to improve rendering speeds from about 2.5 fps to 12 fps using occlusion culling.

Zhang and Turk [227] use occlusion as a visibility metric to reduce LOD in interior regions of single objects, based on image sampling over a bounding sphere.

Govindaraju et al. [76] use two GPUs in an *occlusion switch* for occlusion culling based on hardware occlusion queries for bounding boxes, with a third GPU used for rasterization of visible primitives. This is combined with pre-computed Hierarchical LODs [57].

Only a small number of point based rendering systems incorporate occlusion culling, though existing algorithms for polygon scenes may be applicable or adapted. The Far Voxels system by Gobbetti and Marton [71] uses a BSP tree to construct a vertex tree for LOD, represented as Voxels at branching nodes. The system is a hybrid as it also renders using triangle strips at the leaves of the hierarchy. Occlusion culling is carried out using hardware occlusion queries, with a front to back rendering order. As splatting is used, this approach can also be considered a point based system. The *deferred splatting* from-point technique by Guennebaud, Barthe and Paulin [81] combines level of detail control and occlusion culling in an *elliptical weighted average* (EWA) *surfel* based architecture. Several passes are used to exploit frame coherence. In frame i, the *visibility splatting* technique (see Section 2.9.5.5) rasterizes visible splats $B(i-1)$ in the previous frame. A second pass more cheaply renders all points as indices to a target buffer, with depth buffer tests. This pass establishes a set of visible splats $B(i)$ in frame i. The remaining splats $B(i)-B(i-1)$ are then visibility splatted to include newly visible splats. EWA splatting is then used in a third pass (see Section 2.9.5.6) to rasterize the image. The system is shown to render a forest with 6,800 trees at 10-16 frames per second (fps), each tree containing 750k points. The algorithm still requires the point test pass and as such is not independent of the number of depth layers in the view volume.

A similar from-point temporal coherence algorithm is used in the *layered point cloud* system by Gobbetti and Marton [72] (see Section 2.7.2 and Section 2.9.4.3) in a hierar-

chical point set system, with a point set union accumulation for level of detail. Frame $i-1$'s points are used as occluders in frame i. In a first pass, the point set hierarchy is traversed to obtain a set of candidate point sets for i, only rendering those that were visible in frame $i-1$. In a second pass, icosohedron point set bounding volumes are tested for visibility using the *ARB_Occlusion_Query* hardware extension in OpenGL. Visible candidate point sets not in $i-1$ are then rendered.

A terrain specific occlusion culling system is given by Stamminger and Dretakis [194] based on their procedural grid based $\sqrt{5}$ sampling scheme.

Having looked at occlusion culling techniques, integrated systems with LOD and point based occlusion culling systems, a number of problems are of note. Firstly, unless occlusion culling is combined with LOD control, the LOD system will distribute detail over objects or surfaces that are occluded. A substantial number of occlusion culling systems are only applicable to static scenes due to their level of pre-processing. A number of techniques require occluder pre-selection, which is possibly difficult to achieve efficiently as identification of occluders is in a sense, the visibility problem itself and such systems will not necessarily cull to all occluding geometry. The use of levels of occlusion to drive LOD control [9] is likely to lead to poor results, e.g. when a face partially occludes half a second face at similar distance, there is no reason why half the rear face need be represented in any less resolution than the nearer. Such heuristics are not required if LOD and occlusion culling may be carried out at surface level. Techniques that are intentionally approximate [9] [121] can make substantial errors and these must be limited for a system to be effective.

At the time of writing, no existing point based rendering systems incorporate occlusion culling solutions that take advantage of the point based nature of the underlying scene representation for general scenes. Most recent occlusion culling methods, although efficient, rely on hardware rasterization for occlusion measurement, although exploitation of frame coherence is becoming more common and increases efficiency substantially when coherence is high. Most systems also do not terminate when the image is complete and therefore the complexity of their algorithm is not independent of the number of depth layers in view volume. These aspects are addressed in the algorithm in Chapter 3.

2.8 Image Based Rendering

2.8.1 The Problem

Rendering 3D scenes using 3D geometry as a data source is potentially very costly, even for current hardware. Image Based Rendering (IBR) takes an alternative approach by reusing existing images or parts of images, to construct new images when the viewpoint or scene has changed, as an approach to modelling a scene's 5D Plenoptic function [146] [5]. In the context of 3D rendering, images or image components may be derived from an existing rendering algorithm source as required.

2.8.2 Image Based Rendering Techniques

Image based rendering approaches use and potentially adapt images for rendering, rather than re-generate their appearance by sampling their underlying source in every frame. In this way, they can be totally or partially independent of scene complexity. Surveys on IBR techniques are given by Oliveira [150], Zhang and Chen [225], Shum and Kang [187], Kang [115] and Popescu [159].

The majority of IBR systems address the problem of constructing novel views using existing images taken near to the viewpoint. The most extreme forms of image based rendering include the Lightfield [129] and Lumigraph [74] that store light representing the plenoptic function for fast rendering, at the expense of lengthy pre-processing times, memory use and scene dynamics.

An early use of images of the real world is given by Lippman's [135] videodisc based system for rendering views of a city, based on fetching images of discretized views near the viewpoint from a database on the then new videodisc that offered significant storage possibilities. The popular QuickTime VR and similar systems are also a similar example [27]. Systems that are based purely on discretized views however, may suffer from motion with notable jumps. Subsequently, image based rendering techniques have addressed the problem of applying warping functions to one or more source images to reconstruct novel images from viewpoints that are not in the initial image set. Chen and Williams [28] apply a morphing technique between nearby views that is effective when views are similar, based on knowledge of optical flow. Mark, McMillan and Bishop

[142] also warp multiple reference frames to overcome dis-occlusion problems that cause holes at novel view points.

Texture mapping is an early, simple form of image based rendering, using textures to represent small, higher level of detail geometry cheaply [18]. An image based LOD control technique is given by Cohen, Olano and Manocha [38], that extends the concept of texture mapping, to represent higher resolution geometry appearance on lower level of detail models, using texture and normal maps. Error metrics measure the deviation between the original surface point and the textured approximation that are used to provide error bounds in image space.

Pure IBR is not readily applicable to dynamic 3D rendered scenes, particularly when the geometry is required to be available for other tasks such as dynamic scene updates, collision detection or simulation tasks. Therefore, hybrid approaches have evolved. Latterly, IBR has been applied to rendering algorithms as a speed-up to cache images of selected scene regions, exploiting frame coherence. Typically, the image of one or more objects is texture mapped onto a polygon, with transparency, that is rendered in their place, referred to as an *impostor*. Maciel and Shirley [140] use an octree based approach to cluster the scene into object hierarchies. Texture mapped polygons are then used to represent objects or groups of objects to achieve a consistent frame rate. The authors also combine this with a polygonal level of detail controller similar to that of Funkhouser (see Section 2.3.5). However, the impostors are pre-computed and chosen at run-time and thus is only applicable to static scenes.

Pre-computation of many views of objects, to reduce error, can incur expensive storage overheads. As such, dynamic impostor generation can be used to overcome this issue. Schaufler [177] generates impostors at render time and evaluates their validity using angular error metrics, whilst re-using impostors for as long as possible. Texture resolutions are chosen based on the impostor's context and thus also do not incur overheads of long term storage at a high resolution. Similarly, Shade and colleagues [185] take a dynamic caching approach based on spatially partitioning the scene using a BSP tree, to permit back to front rendering for correct compositing of transparency in the frame buffer. Each internal node in the BSP tree can be assigned an impostor that represents its child trees, texture mapped onto a rectangular polygon through the node's geometric

center. Error metrics evaluate the life expectancy of creating a cache at a node of the BSP tree. Additional error metrics then validate cached images by assessing the difference between the impostor and its rendered geometry. Caching is hierarchical in that impostors lower down in the tree may be used in the creation of those higher in the tree.

Impostors can have a number of problems, including lack of parallax, notable popping effects when switching between impostor and geometry and coherence problems where objects should be close or intersect. The addition of depth information in cached images, such as with Schaufler's *nailboards* [178], Shade's *depth image* [184] or *depth meshes* used in the MMR system by Aliaga [7], can reduce error by introducing motion parallax, where source geometry is close or intersecting. Shade and colleagues use Layered Depth Images (LDI) [184] that store multiple surface depths per pixel to alleviate potential disocclusion errors that can prevail when surfaces that were hidden when the impostor was generated, should come into view, but do not, due to parallax. Layered depth images begin to overlap with some point based rendering approaches in that they can use warping and splatting techniques [184]. Shade and colleagues [184] achieve several frames per second on relatively low end hardware. Wimma, Wonka and Sillion [216] also use LDIs in a point based impostor scheme with anti-aliasing, based on view cells for static scenes.

As seen in this overview, impostor techniques are incrementally advancing towards point based rendering methods, in order to maintain additional efficiency, whilst reducing errors in their approximation. Point based caching techniques are often readily applicable to point based architectures. The next and last section in this chapter examines point based rendering in depth.

2.9 Point Based Rendering

2.9.1 Introduction to Point Based Rendering

The algorithm developed in Chapter 3 onward, can be considered a point based rendering technique. This final section of this chapter looks at aspects specific to point based rendering (PBR). The vast majority of realtime rendering systems are based on a polygon rendering algorithms (see Section 1.3) and the standard rendering pipeline (see Section 1.2). The inefficiencies of the standard polygonal pipeline are discussed in Section 1.4. The most notable point is that as the complexity of 3D environments increases, afforded by ever faster hardware and optimizations, the typical size of a polygon is either a few pixels in area or quite possibly substantially smaller than a pixel such that it shares the pixel with other polygons.

At some point in time, many visualization and gaming environments will be able to provide unique geometric detail for every pixel, rather than the area coverage that polygons have offered in the past. As scene detail, memory availability and procedural environment generation methods mature, it is unlikely that polygonal area coverage will be required.

Therefore, the use of polygon representations for the purposes of image generation may come into question. It is unlikely that their use will be more efficient than architectures specifically designed to generate images in this new context. This does not mean, however, that polygonal representations will not be required for other purposes such as CAD/CAM, finite element methods or other applications requiring computational geometry tasks for analysis or manufacturing.

Point Based Rendering is a field with an associated class of algorithms which have recognized the inefficiencies and difficulties of using polygons and instead, have chosen to use points or more complex surface elements as a rendering primitive. PBR algorithms typically attempt to efficiently sample and construct the image.

The 3D surfaces of an environment are sampled in a specific manner and are stored efficiently. The system then makes decisions at runtime about which surface samples to use

to sufficiently sample the image. Most algorithms make provisions to handle surface and image sampling densities such that holes to not appear.

The use of points was addressed early in 1979 by Csuri [46] and a treatment for high quality rendering using displacement maps given by Levoy and Whitted [130]. The volume rendering and particle system fields also have applicable literature due to similarities. Interest in PBR has grown since the publication of QSplat by Rusinkiewicz and Levoy [171] [172] and the Surfels technique by Pfister and colleagues [158] at SIGGRAPH 2000, closely followed by the Randomized Z-Buffer by Wand and colleagues [205] at SIGGRAPH 2001. By the year 2006, there are probably approaching a hundred PBR publications with a wide range of rendering engines, primarily for desktop systems and a dedicated annual IEEE/Eurographics symposium on point based graphics. A PBR system has even been developed for a mobile device by Duguet and Drettakis [52]. Due to the success and future potential demonstrated by existing PBR systems, it is likely that research in this area will continue for some time and it would seem likely that direct hardware support may follow.

A number of other research areas overlap with some aspects of PBR. Particle systems are an earlier incarnation generally used to represent non rigid forms with specific dynamical behaviour, such as fire, water [167], snow [59], smoke [46] and clouds [92]. Most of the emphasis has been placed on the dynamic behaviour and inter-relationships of particles. Volume rendering is another field which overlaps with PBR where concept of *splatting,* also used in PBR, was first introduced by Westover [211].

Point based scene representations have also recently been used for fast ray tracing by Adamson and Alexa [4], Adams and colleagues [2] and Wald and Seidel [204] and for radiosity by Dobashi, Yamamoto and Nishita [51], but only projective, local illumination approaches are considered here.

The following sections describe the benefits and technical aspects of PBR systems, such as common approaches, data structures and rasterization techniques.

2.9.2 Benefits of Point Based Rendering

Point based systems have a number of specific advantages. The desire for very high scene and image complexity suggests that ultimately, pixel or sub-pixel sized primitives are required in favour of area based primitives such as polygons. Discarding connectivity reduces storage overheads, processing and the need for careful topological considerations and operations, e.g. for LOD. Point based systems can represent arbitrary topologies easily. They can also remove some requirements for surface parameterizations. For example, if per-point colour is used instead of texture maps, pre-computed surface shading, commonly termed *texture baking,* can be carried out without requirements for unique one to one mappings between surface points and texels.

Level of detail control can be a simpler task, one of ensuring sufficient image sampling in an *output sensitive* architecture, limited by screen resolution. Future rasterization hardware specific to PBR may be simpler and faster, due to a more basic set of core operations. For example, perspective correct texture mapping and mip mapping may not be necessary. It may also be feasible to implement the processing of PBR data structures in hardware.

A common assumption of point based approaches is that holes may appear if surfaces are insufficiently sampled in object space, or their representation is insufficiently sampled in image space. Solutions are to detect and fill holes or simply ensure that sufficient data is available for required views. Instead of attempting to interpolate missing object space data in image space like polygon methods, future systems may address how object space data can be procedurally generated or interpolated on demand. Were sufficient object space samples unavailable for image space sampling, algorithms can still ensure that holes do not appear in surfaces using techniques such as splatting.

With these aspects in mind, the next section will examine high level concepts and components which are common to most PBR algorithms.

2.9.3 Common Aspects of PBR Architectures

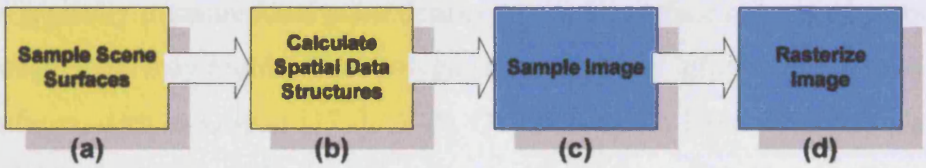
The components required by PBR systems are sometimes common because they need to solve similar problems and often do so in similar ways.

PBR systems that use zero volume points for scene representation typically project them into the image as pixel sized primitives. Those that are area or volumetric in object space area typically projected to an image space equivalent.

To provide contiguous surfaces without holes, systems either use very high densities that ensure coverage, represent the object space surface in a contiguous form to project to a contiguous representation, or find a way of filling holes in the image.

A high level block diagram of processes in a generic system is shown in Figure 18. Initially, objects are sampled from their source form, typically from point clouds, polygonal or spline representations. At this stage, the system will extract as much information as it requires, including texture sampling in some cases. The resulting sets of samples are then processed into containing data structures, typically based on hierarchical spatial methods such as K-D Trees, Octrees, point clouds or principle component analysis (PCA) based clustering.

FIGURE 18. **Common PBR processes. (a-b) are pre-processes, (c-d) are runtime.**



These initial stages are usually performed off-line on current hardware, but future hardware and methods may enable runtime operations. Stages (c) and (d) are runtime stages. The data structures created in (b) are sampled and a subset passed to (d) for rasterization. Most existing algorithms are not quite capable of fully sampling all pixels in an image, so some degree of image reconstruction is required in the rasterization stage (d), usually a splatting technique or EWA filtering (see Section 2.9.5).

2.9.4 Point Based Rendering Methodologies

This section takes a more detailed look at specific problems and how various existing PBR systems solve them. These sections examine scene sampling, data structures and rendering processes of PBR systems, including procedural, randomized and hybrid techniques.

2.9.4.1 Scene Sampling

Virtual environments are typically designed in modelling or CAD packages using polygonal or spline based techniques. More recently, high resolution geometrical modelling systems such as Z-Brush[1] and point based systems such as Pointshop 3D [230] are available. Alternatively, models can be scanned, resulting in millions of point samples or generated procedurally (see Section 2.9.4.5). To create a PBR system, point samples must be extracted to represent surfaces. Normally they are extracted as a pre-process, but some methods such as the Randomized Z-Buffer system by Wand and colleagues [205] use pre-processed sample density based data structures to sample polygonal surfaces in real-time. Procedural methods such as those by Stamminger and Drettakis [194] also sample functions at runtime.

In some cases, the surfaces can be sampled independently of the data structures used to store them, though some methods such as that of Levoy and Whitted [130] and Surfels [158] are more strongly linked to their spatial data structures.

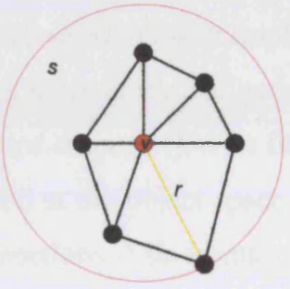
Systems typically measure local point density to ensure surface coverage by point samples, though some may require local polygonal connectivity information to form contiguous surfaces, such as QSplat [171], [172]. QSplat takes an input mesh and places one spherical sample at each of the mesh's vertices with unique radius, to guarantee a contiguous surface through the union of spheres. The resolution and distribution of the surface point sampling is therefore a function of the mesh's vertex distribution, relying on the data source being highly detailed. Geometric attributes extracted at each vertex are *position*, *radius* and *surface normal*. The radius is given by the maximum polygon edge length of all polygons defined by the vertex. This serves to guarantee contiguous connectivity between samples because the approximating sphere is a super set of the space defined by the polygon patch it represents. If the original mesh has no holes, then neither will the point sampled representation. Figure 19 shows an example vertex v defining sample sphere s, centered at v. The radius is defined by longest edge r to conservatively bound the patch defined by v's local polygons. Vertex normals can be computed con-

1. (c) 2006 Pixologic

ventionally as the mean of the surrounding polygon normals. Colour can be extracted simply as a per-vertex colour if available, or sampled from v's texture coordinate.

Although this approach is also optionally available in the *Canopy* system (see Chapter 3) it is limited to high resolution models and not suitable for lower resolution models, such as those designed by hand. This would lead to samples that are too large and textures would not be sufficiently sampled to be represented by point sample colouration. In addition, it can result in a mixture of sample sizes, that may be less applicable to simple spatial partitioning without splitting.

FIGURE 19. Vertex *V* sample sphere defined by largest extent of local mesh area

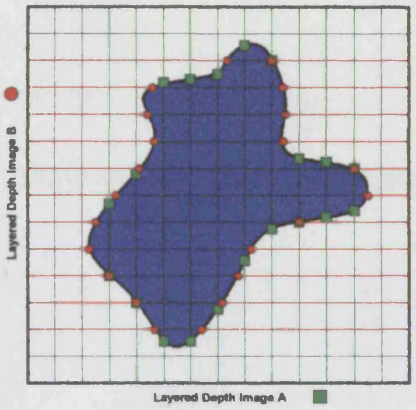


Tobor, Schlick and Grisoni [202] sample given polygonal models on a regular grid using voxelization techniques they term *surfelization*. A similar method is used by the *Canopy* algorithm (see Chapter 3). This method is likely to be faster than similar techniques such as ray casted intersection methods. Voxelization requires an existing surface definition, but is independent of the resolution of the input model and therefore is not subject to the issues of vertex patch sampling.

The Surfels method of Pfister and colleagues [158] samples an object using three orthographic projections. Ray casting is used to derive three top resolution *layered depth images* (LDI). A plan view showing a 2D construction is shown in Figure 20, which results in two LDIs A and B. The 3D form creates a *layered depth cube* (LDC).

As an optional optimization, local samples can be grouped from each axis to the grid intersection points, offering a maximum reduction of 3 to 1. This results in a single LDI which offers speed-ups in block warping methods used for perspective projection.

FIGURE 20. 2D view of orthographically sampled object with 2 LDIs (A & B)



This system forms the highest resolution of an octree hierarchy in [158] where these samples are represented and sub-sampled in lower resolutions, higher up the octree. Each lower resolution stores a subset of those in the leaf nodes of its child trees. Colour is pre-filtered using an *elliptical weighted average* (EWA) filter. This involves projecting tangent discs positioned and oriented at the object space sample points into texture space using texture parameterization functions. This forms a system of overlapping ellipses in texture space which are used to filter the colour at the object space samples. To reduce texture aliasing at runtime, several colour samples are made at different ellipse scales in texture space per surfel, resulting in a system similar to *mipmapping*, which the authors term *surfel-mipmapping*. A similar orthographic sampling method is used by Grossman [79] [80] where an equilateral triangle sampling distribution is used in the sampling plane to reduce the number of samples used for a surface, in contrast to a regular grid.

The hybrid point and polygon rendering system presented by Cohen, Aliaga and Zhang [37] uses a system of replacing polygons with an hierarchy of optimized point samples at opportune times during rendering. Polygons in models are sampled directly as a pre-process. A triangle is sampled in rows, similar to a rasterization process. Points are represented using spheres in the implementation, but during sampling are approximated using flat non overlapping squares on the polygon plane as they are more easily tiled. These squares define a series of overlapping spheres which are guaranteed to cover the polygon conservatively, with no holes. For squares of dimension (width and height) d, the sphere radius can be obtained simply by:

$$r = \frac{d}{\sqrt{2}} \quad \text{(EQ 4)}$$

The authors derive a function $d = f(A, s)$ to calculate a sampling distance d such that no more than s samples are used to cover a triangle with surface area A. The theoretical minimum is given by:

$$A = sd^2_{min} \Rightarrow d_{min} = \sqrt{\frac{A}{s}} \quad \text{(EQ 5)}$$

The actual sampling distance required for a specific triangle is found numerically using d_{min} as a lower bound. An upper bound is formed by successively doubling d_{min} until the number of samples is greater than s. The search is then performed within these bounds to a specified error tolerance. These methods are then used as components for a Benefit/ Cost optimization for d.

The Differential Point (DP) Rendering system of Kalaiah and Varshney [113] performs a local regional surface analysis of a differentiable surface, resulting in area based splats which are approximated by polygonal surfaces exhibiting normal behaviour similar to that of the local surface curvature in the vicinity of the sample. The system requires surfaces that are easily analysed for second degree curvature. Parametric surfaces lend themselves well to this kind of task due to their bi-variate nature. With substantial effort, spline surfaces can be fitted to existing polygonal models. The authors sample the parametric domain of NURBS surfaces uniformly to obtain point samples. Because each surface sample is capable of representing more complex behaviour of an area of the local surface, the system is claimed to require fewer samples than other PBR systems because it is able to remove redundant samples which are well approximated by their neighbours.

Some methods are primarily concerned with the analysis of point cloud data for multi-resolution model calculation (see Section 2.3.3) resampling, reconstruction and interactive editing. The Pointshop 3D system [230] has been specifically designed for sculpting and carving using normal displacements with cleaning and resampling facilities, also concentrating on user defined parameterizations for applying textures. Other modelling techniques also include [86] [87] and [156].

Further work has been concerned with the automatic recognition of discrete features in models. Gumold, Wang and MacLeod [85] process raw point clouds without prior surface reconstruction. A neighbouring sample graph method is used, based on feature membership weighting scheme. The Principle Components Analysis (PCA) technique is fast finding itself applications in point based rendering as it is readily applicable. Pauly, Keiser and Gross [155] extract or reconstruct based on local PCA methods for feature classification.

Sampling and scene creation can also be carried out at render time using randomized (see Section 2.9.4.2) or procedural techniques (see Section 2.9.4.5).

2.9.4.2 Randomized Scene Sampling

The process of constructing an image results in a sampling of the scene, mapped onto pixels in some way. Ideally, sampling would only address points that are guaranteed to contribute to pixel colours in the image.

Few rendering algorithms rely on probabilistic methods, possibly as they incur a risk of rendering images with artifacts. The Randomized Z-Buffer algorithm by Wand and colleagues [205] is an example of PBR which followed this approach with positive results. The method can be considered *hybrid* in that it renders polygons if they have a sufficiently large image space area, rather than point sampling them as this becomes more efficient. The authors claim that the method will never be slower than conventional polygonal z-buffer rendering and that image quality is comparable. Sampling is guided by image space area classifications of triangles in the scene, which are still maintained for sampling at render time, unlike other PBR algorithms. Image space area classifications and depth bounds are made possible using an octree spatial partitioning, which is capable of undergoing dynamic changes at runtime, e.g. object insertion, deletion or motion. Each octree node's volume stores polygons which can be contained by it, but not the volumes of its child octants, classifying polygons based on size. Orientation is ignored. For a given viewpoint, the algorithm selects boxes associated with nodes of the octree to take part in the rendering, based on view frustum visibility and a depth control mechanism based on the ratio of the nearest and furthest scene surfaces constrained by a threshold. In this sense, the algorithm is hierarchical in that it traverses an octree from

the root in every frame, though it does not use any form of level of detail control based on this hierarchy. All image sampling is sourced directly from the high resolution model geometry provided. The aim of the algorithm is to point sample all object surfaces in the image such that they are uniformly and sufficiently sampled in image space, potentially with complete pixel coverage. Distribution of the number of samples between objects is dependent on their projected area, classified by the octree groupings. If specific polygons have a large image space area, they are passed to a conventional rendering pipeline as an optimization. Otherwise, points are sampled which are guaranteed lie within the triangle being sampled. Sampling is performed such that it conforms to a distribution dictated by the relative areas of triangles. Each octree node has a distribution list to enable sample surface selection at render time. Such lists are formed by a hierarchy in the octree, where a node's list is the concatenation of those of its child nodes and those according to the triangles stored in the node's own volume. The system attempts to sample the scene such that every pixel in the image has a very high probability of receiving a point sample from the closest surface. This is formulated in terms of the classic *occupancy* problem, where a number of balls n are chosen such that if thrown randomly and independently into v bins, there is a specifiable probability p that each bin has received a ball, if $n \geq v$. This models the probabilistic projection of visible, front surfaces into screen space. The authors then account for the occluded surfaces in this sampling scheme by adding extra pixels to be filled to cater for occluded surfaces. A relaxed form is also described using splats for faster frame rates. The Randomized Z-Buffer however, is stated to require between 11 and 14 over samples per pixel, which is high.

A caching scheme is also used which reuses sampled points if the sampling density for a group of polygons does not change substantially between frames. This appears to yield speed-ups of up to a magnitude. Multiple, separate caches are maintained, associated with different regions of the scene, associated with octree volumes. Sample caches are deleted based on a *least recently used* (LRU) scheme common to resource management algorithms.

Scenes are represented using scene graphs, similar to those of various other APIs such as Open Inventor, Performer[1] and Java3D[2]. Instancing provides an efficient method for the creation of extremely large scenes. One of the scenes tested by the authors contains 10^{14}

triangles, which is rendered at several frames per second. Speed-ups in general are reported to vary from a few to many orders of magnitude when compared to polygonal z-buffer rendering of the same scenes, using an 800MHz AMD Athlon CPU and nVidia GeForce 2 GTS graphics card.

The system presented by Stamminger and Drettakis [194] also samples randomly over parametric surfaces defined by a bi-variate function $f(u, v)$ using pseudo random Halton sequences for a more even distribution than most random number generators. A second sampling distribution is used for procedurally generated surfaces, based on a method referred to as $\sqrt{5}$ sampling which is a hierarchical, selective refinement method. Given an initial uniform grid of samples with sample distance h, the method defines successively higher resolution grids with each higher resolution containing all previous lower resolutions as a subset. The new grid is rotated by $\text{atan}(1/2)$ and has sample distance $h' = h/(\sqrt{5})$. The grid can be refined selectively if required, spawning four new points from each original point. The result is a fractal distribution which does not fill the whole original area, unless samples outside the boundary are included and are allowed to rotate samples into the initial grid area under consideration at higher resolutions.

Levoy and Whitted [130] also use a randomized point ordering for rasterization using a binning scheme for blending and hidden surface removal. Randomized sampling is again used by Kalaiah and Varshney [114] to refine lower resolution models resulting from PCA compression, based on Gaussian distributions defined by the PCA results. The authors report that several orders of magnitude compression can be obtained.

2.9.4.3 List, Octree & K-D Tree Scene Representation and Rendering

Having sampled a scene using whatever methods an algorithm uses, the point sample data must be structured in some way to support rendering tasks. It may be that the sampling method itself dictates, or strongly suggests the nature of these data structures, as in the case of Surfels [158], which samples and provides lower resolution models using an octree. This section will look at flat list, octree [109] and k-d tree [15] based algorithms.

1. (c) SGI
2. (c) Sun Microsystems

The simplest form of spatial data structure to support point sampled environments is to store them in a list in an arbitrary, random order or grouped according to the surfaces they belong to. The Differential Point rendering system [113] works in this way because it uses a pre-defined set of efficient surface samples which approximate large areas with high accuracy, using conventional polygonal rendering with a z-buffer for output.

Tobor, Schlick, Grisoni [202] store a regular grid of surfels using a hash table in a data structure they term a *surfel collector*. No levels of detail are currently provided by this method.

Levoy and Whitted [130] in an early work on point based rendering, define a displacement map of point samples from a 2D grid. The approach concentrates on anti-aliased filtering with opacity. Image space sample density is evaluated by projecting an object space tangent plane positioned at a sample. The density is inversely proportional to the area of the parallelogram formed by the projection. Pixel filtering is used, positioning Gaussian filters at pixel centers. The point density evaluation is used to normalize the weighted average contributions. For rasterization, points are placed into bins representing contiguous surfaces. Point contributions in each bin are blended. Point membership in bins is based on thresholded depth tests. Overlapping bins are merged. The approach does not address scalability issues beyond surface rasterization.

Grossman and Dally [80] orthographically sample an object using an equilateral triangulation distribution to their sampling which is claimed to use less samples than a regular grid. The sampling density is increased to account for polygons oblique to the orthographic projection planes. Samples are split into blocks. These blocks are visibility tested using view frustum culling and back face culling based on visibility cones (see Section 2.6). Visibility masks based on that of Zhang and Hoff [226] are used to occlusion cull regions interior to the object when viewed outside the convex hull. Points in blocks are projected to image space using a fast incremental warping. Holes in the image are then identified and filled using a push-pull system based on a hierarchical image pyramid (see Section 2.9.5.4) with shading in between the pull and push stages. Using a 333MHz CPU, 256x256 images are rendered at between 0.077s and 0.282 seconds.

Most PBR algorithms use multi-resolution hierarchies that are refined at render time. Selective refinement has been discussed in the context of level of detail control in Section 2.3.5. Selective refinement is a multi-resolution technique which is readily applicable to runtime image construction for PBR systems using hierarchies of point samples. Two common types are systems that accumulate samples hierarchically, taking a union of sample sets and those that replace points with a larger number of points (see Section 2.3.2). Hierarchies allow amortized metrics and decisions to be applied to a lower resolution representation, on behalf of a larger number of higher resolution child points for increased efficiency. In particular, view volume culling, back face culling and level of detail are commonly achieved this way in PBR systems, though hierarchical occlusion culling is so far rare in the literature, specifically for PBR scene representations. A specific multi-resolution image solution is often defined as a subset of nodes in the hierarchy, where given a node in the solution, no other node below in the hierarchy, is also in the solution. This solution may be derived from the root in each frame, or frame coherence algorithms may reuse the solution to adapt it from one frame to the next, as seen in occlusion culling algorithms (see Section 2.7).

In 1976, Clark outlined the benefits of sphere hierarchies for view volume culling, level of detail, occlusion culling, frame coherence, output sensitive rendering limited by image resolution and working set management for out of core rendering. In this concept, a bounding sphere hierarchy is used for polygon or patch surface representations. Twenty to thirty years later, many systems are developing techniques similar to this approach. Polygon based selective refinement systems were initially developed, including Hoppe [100] [99], Lau and Green [126], Luebke and Erikson [136] and Xia and Varshney [221] (see Section 2.3.5).

The QSplat system by Rusinkiewicz and Levoy [171] [172] uses a tree hierarchy of bounding spheres to represent a scene, the highest resolution samples having been acquired using the vertex patch sampling process described in Section 2.9.4.1, shown in Figure 19. A *k-d tree* is constructed to spatially partition using axis aligned planes, chosen such that each partitions along the longest extent of the bounding box of the samples to be partitioned. This helps maintain an equilibrium with respect to the aspect ratios of the volumes down the tree. At each node of the tree, an approximating sample is created

which bounds the samples of the child nodes. Colour and normal attributes are also approximated. To reduce storage overheads, the resulting binary tree nodes are grouped to form a quarternary tree (1 to 4 branching) eliminating two intermediate nodes. Therefore each branching node's sample must approximate four child samples. Each node's position and radius is delta encoded using lossy quantization for reduced storage and streaming [172]. The tree is traversed from the root in each frame, driven by a selective refinement algorithm. Splatting is used for rasterization. Pseudo code for a simplified version of QSplat's main loop is shown in Figure 21 which has some minor similarities with the *Canopy* algorithm (see Chapter 3). During traversal there are a set of nodes considered active, undergoing refinement. Any one of these nodes can be chosen for refinement. Given a node, QSplat first sees if the node can be culled based on the view frustum or hierarchical back face culling. If not visible, no child nodes are considered further during rendering. If the node is visible and is a leaf, no more detail is available and a splat is rendered. If the node is not a leaf, it undergoes a benefit evaluation. If the algorithm considers the benefit of recursing further to be too low, a splat of the node is drawn and the child trees are not traversed. Otherwise, the child nodes are traversed and the they replace the parent node in the current image solution. Benefit is evaluated based on image space size and target frame rates, with the image space size threshold adjusted between frames based on performance achieved, forming a reactive feedback LOD control system (see Section 2.3.5). On a 366MHz system, the authors achieve about 5fps and traverse 250k to 400k nodes per second, drawing 50k to 70k splats per frame.

FIGURE 21. Simplified hierarchical PBR refinement used in QSplat

```
TraverseHierarchy(node)
{
        if (node not visible)
        {
                skip this branch of the tree
        }
        else if (node is a leaf node)
        {
                draw a splat
        }
        else if (benefit of recursing further is too low)
        {
                draw a splat
        }
        else
        {
                for each child in children(node)
                {
                        TraverseHierarchy(child)
                }
        }
}
```

Woolley, Luebke and Watson's Interruptible Rendering [218] defines a different refinement for LOD control with a QSplat style hierarchy based on trade offs between spatial

error imposed by LOD and temporal error imposed by not rendering immediately. When a temporal error metric exceeds a spatial error metric, rendering is terminated. Their temporal error includes the tracking of dynamic bounding boxes. This results in low LODs and high frame rates at dynamic periods and high LODs at lower frame rates when more static.

The QSplat system is reused in the POP point/polygon hybrid by Chen and Nguygen [29], who store triangles at leaf nodes and use spheres at branching nodes, to alleviate inefficiencies of PBR with large flat surfaces. Several frames per second are achieved for a scene with 250k polygons.

Octrees offer a fast spatial analysis with local clustering, but can suffer from sampling issues if grouping is not considered between neighbouring octants over partitions. The rigidity of the octree representation with its requirement for non overlapping octants, can limit octree based techniques to static scenes, unless efficient rebuild strategies are introduced [196]. Many systems use octrees to structure their sample data.

Whilst not strictly a point based rendering system, Chamberlain and colleagues [26] use an octree as a multi-resolution hierarchy to replace polygonal geometry at leaf octants by rendering octant cells. Faces of each octant are coloured by analysing a view through the face of the contained geometry. Traversing the octree, if the project size of an octant is small enough, the octant is rendered. If the leaves are reached, polygon geometry is rendered.

The Surfels system [158] selects subsets of the leaf node samples for lower resolution representations in interior nodes of an octree, with texture pre-filtering. Rendered blocks are specified by a selective refinement LOD control system. Image space sample density is estimated based on projection of a maximum block sample distance, to predict the number of surfels projected to pixel reconstruction filters, providing a parameter that trades off between performance and image quality. Visibility splatting (see Section 2.9.5.5) is used to identify visible surfaces and holes in the surface coverage that are filled (see Section 2.9.5.4) with Elliptical Weighted Average (EWA) filtering for rasterization (see Section 2.9.5.6). When rendering a 1024x1024 image, frame rates of about

1fps are achieved for objects that originally consist of about 120k polygons on a 700MHz Pentium 3 system, with a throughput of about 250k surfels per second.

The surfels approach is again used by Guennebaud and Paulin [84] in a GPU implementation that incorporates per fragment depth correction. The authors achieve a higher 9fps for an object of 400k surfels, yielding a throughput of about 4M surfels per second. Coconu and Hege [36] use a system similar to Surfels, although a back to front rendering order is included to support transparent surfaces, in place of the *visibility splatting* technique used in Surfels [158]. The system is a hybrid renderer (see Section 2.9.4.6).

Laur and Hanrahan [127] create a hierarchical splatting system for volume rendering using octrees in a selective refinement process based on image space error, rendering and compositing splats in a back to front order in the view volume.

Guthe and colleagues [89] use an octree to formulate a hierarchical wavelet representation of data sampled from a regular grid. The wavelet representation is refined during rendering based on spectral frequencies in a view dependent way. Adams and Dutré [1] also use octrees to perform containment tests for boolean operations.

Botsch, Wiratanaya and Kobbelt [22] use an octree structure to efficient encode points by adding bits to increase precision with decreasing scale, achieving 2 bits per position. Normal vector compression is also included based on an octahedron encoding. A model with 2M splats is shown to render at 2fps.

The Sequential Point Tree by Dachsbacher, Vogelgsang and Stamminger [48] is a hybrid, GPU centric architecture that flattens and stores point hierarchies in graphics memory to cache and reduce bandwidth between the CPU and GPU. An octree of points is used with bounding volumes at interior branching nodes (see Section 2.9.4.6). Guennebaud and Paulin [81] implement a deferred splatting technique that is data structure independent and suited to the sequential point tree technique in [48]. The technique combines view volume, back face and hardware occlusion based culling with frame coherence (see Section 2.7).

The Randomized Z-Buffer method [205] stores polygons in octree volumes to enable classification based on potential area under projection and to place a bound on distance during traversal. A viewpoint dependent, randomized surface sampling technique is used with probabilistic image coverage (see Section 2.9.4.2).

Guthe et al. [88] use an octree to define spatial bounds on simplification in a hybrid system that replaces small polygons with points, also incorporating shadow mapping.

Hopf, Luttenberger and Ertl [98] use a hierarchical representation based on octrees or principle component analysis (PCA) that is better suited to the data set cluster shape. A selective refinement controls LOD, based on image space size for image quality and performance trade-offs. Points are sorted for rasterization of non commutative blending for transparency. OpenGL points are used for rendering splats. The system renders about 4.5M splats per second without sorting and 3M splats with sorting on an Intel Pentium 4 2.8GHz system and nVidia GeForce FX5800.

Kalaiah and Varshney [114] use octrees as a multi-resolution data structure for localising principle components analysis (PCA) on point based models. Their method analyzes orientation frame, mean and variance information for localized cells. Unusually, their method does not attempt to fully reconstruct higher resolution information, but reduces it to a compressed form, e.g. for efficient storage and transmission, subsequently using random Gaussian probability distributions based on the PCA to refine for view dependent rendering.

Guennebaud and Paulin [83] store points in an octree structure and interpolate local regions procedurally using Bezier patches to increase the sampling density for close views (see Section 2.9.4.5). Pajarola [151] uses a *point octree*, adaptively adjust to data, where the central split vertex formed by the intersection of partition planes in the octant, is the mean of the contained points. A blended procedural interpolation scheme is used (see Section 2.9.4.5). Mantler and Fuhrmann [141] import high resolution microscopy data into a grid format and construct a hierarchy of overlapping blocks, similar to an octree for LOD in a hybrid system (see Section 2.9.4.6).

A hierarchical point cloud is used by Gobbetti and Marton [72] in one of few PBR systems that combines hierarchical LOD, view volume culling, back face culling and hardware based occlusion culling (see Section 2.7.2). A top down partitioning scheme partitions the largest axis of the bounding box in a k-d tree style data structure. Point cloud unions are taken to progressively increase the image sample density in a selective refinement LOD control system. Each clouds consists of a few thousand point samples. LOD decisions are lower cost than most systems, due to the amortization over all points in each set. Two LOD control approaches are used, one based on quality by specifying required image splat sizes and one based on performance that hierarchically adds points as a cost metric, terminating traversal when a budget is exceeded. In a first traversal stage, a set of candidate point sets is identified, whilst rendering those that were visible in the previous frame, forming a frame coherence algorithm. In a second stage, the hierarchy is traversed with point set icosohedron bounding volume occlusion tests to establish visibility, using the hardware based *ARB_Occlusion_Query* extension in OpenGL. If a newly visible set exceeds a visibility threshold, the point set is added to the occluder list for the next frame. If a member of the list is not visible, it must be removed. A third stage then renders point sets that were not rendered in the first stage. Data is delta compressed using the Lempel Ziv algorithm for streaming. For large data sets, out of core rendering is employed to fetch from disk. The has a splat rasterization throughput of 40M splats per second using OpenGL points. A model with 24M points is shown to render at 5 fps with occlusion culling reducing this to 6.3M splats for rasterization.

Far Voxels by Gobbetti and Marton [71] uses a K-D tree style BSP tree to partition polygonal models that are re-processed to clusters in a second pass, with voxels at interior nodes. This system incorporates both LOD and occlusion culling.

2.9.4.4 Tree and Alternative Scene Representation and Rendering

Various PBR systems use alternative scene representations that will be examined here. These tend to include more complex surface analysis, hierarchies, refinement or point representations, other than the flat list, octree or k-d tree hierarchies examined in the last section.

Stamminger and Dretakis use regular grid sampling with a $\sqrt{5}$ refinement scheme that selectively increases resolution to a higher level grid in a rotated alignment that supports procedural generation (see Section 2.9.4.5).

Xu and Chen [222] construct a hierarchy from interactively and semantically segmented scanned range images of urban areas with colour, to improve visibility culling. Points are stored at leaf nodes, with interior nodes grouping discrete surfaces or objects. Visibility splatting and point sprite gaussian splatting are used for rasterization. For an 800x600 image, a scene with 1.3M points is rendered at about 16 fps on an Intel 800MHz Pentium 3 with nVidia GeForce Ti4600, realizing a splat throughput of about 8M per second. This approach may suffer if spatial groupings are not efficiently tight and balanced in comparison to analytical approaches.

Similar to the progressive mesh edge collapse decimation by Hoppe [100] [99] [102] (see Section 2.3.2 and Section 2.3.3) a greedy splat decimation system is given by Wu, Zhang and Kobbelt [220] to improve point based representations at lower LODs, based on ellipses. Each point's k nearest neighbours are pre-computed, with a least squares plane to support a new sample. A graph of potential collapses is formed and a priority queue of splat merge operators is maintained. The least squares metric assesses a new point's approximation to its neighbourhood. Cohen, Aliaga and Zhang [37] define a hybrid system using a graph of simplification operations for hierarchical LOD (see Section 2.9.4.6).

In contrast to regular scene representation methods, Krüger, Schneider and Westermann's Duodecim system [124] uses a hexagonal close packing grid of cells to contain point samples from scans, with multiple grid resolutions for LOD. Connectivity between filled cells is used to establish surface relationships. Run sequences are defined around filled regions as a rendering traversal order. Data in runs can then be 3-bit delta encoded for reduced storage, denoting traversal to one of 6 neighbouring cells, with a 5-bit encoding for surface normals. This data is encoded into textures as input for GPU processing. At runtime, the required run for LODs are transferred to the GPU. A 150M point model is reduced from 10GB to 230MB of memory, but with a long encoding time of 10 hours. A 2.5M point model is rendered at 16 fps with an encoding time of 5 mins.

Fleishman and Cohen-Or define a Progressive Point Set Surface [62] using a multi-level displacement map system. Surface points are projected to local polynomial surfaces using offsets to provide lower, smoother levels of detail and vice versa. A moving least squares method (MLS) is used to define lower resolutions.

Pauly and Gross [153] also define displacement fields in local regions using tessellated patches associated with planes. Local spectral analysis and signal processing functions such as windowed Fourier transforms can then be applied for filtering and resampling tasks.

A more unique approach is also given by Clarenz, Rumpf and Telea [34] who represent point based sampling with finite element based partial differential equations.

Point based rendering algorithms in general need high sampling densities to portray high frequency surface components which parallax. An alternative method has been presented by Kalaiah and Varshney [113], which introduces a *differential point* (DP) representation. This approach attempts to capture and represent greater detail in a set of larger samples. DPs represent rectangular patches of the original surface. Curvature information is stored within the sample, requiring the source surface to be locally differentiable. The authors use source models based on NURBS (non uniform rational B-Splines) because they are easily differentiated at a given point. Surfaces are sampled to DPs uniformly in parameter space as a pre-process. Rectangle dimensions are chosen based on curvature metrics. Generally, more highly curved regions will have smaller rectangles. A second simplification process successively removes rectangles which overlap, subject to the constraint that their neighbouring rectangles fully cover the DP removed. The planar rectangles r_p for each DP are used during rendering to approximate the local curved surface. These rectangles can then be rendered conventionally, using bump mapping hardware. The authors report performance in the region of several frames per second with tens of thousands of point samples. The DP method relies substantially on an abstraction to hardware supported polygonal rendering, which is convenient for practical implementation. However, if hardware support were offered to tasks required by other PBR algorithms, the advantage of additional speed through reduced sample density may not be realized.

Where many PBR systems are concerned with object space sampling to support image based sampling, the edge and point image (EPI) technique by Bala, Walter and Greenberg [11] encodes surface silhouette edge patterns and sparsely point sampled interiors in an intermediate EPI image space data structure. A polygon based scene representation is still used. Silhouettes are found using a hierarchical edge and normal silhouette detection algorithm. Occluded edges are removed based on depth tests. Ray casting is used to establish surface samples. Table based interpolation is then used for surface and attribute reconstruction for rendering. Discontinuities are preserved using a *reachability* system concerned with not using interpolation sources over discontinuities, including shadow boundaries. A render cache is used to store coloured surface samples from the previous image, re-projecting them from the next viewpoint, forming a temporal coherence algorithm. A scene with 250k polygons is rendered at 10 fps with a glossy shaded 512x512 image, where only 20% of the pixels are sampled from the scene. The authors claim a mean speedup of 20 to 60 times per pixel over a full ray casting approach.

2.9.4.5 Procedural Scene Generation and Rendering

Point based methods in general, lend themselves well to procedurally generated environments, traditional forms being simple shapes, fractals, L-Systems [157], parametric surfaces or more complex scene creation approaches. This section will discuss a number of point based procedural systems. These systems require generator functions and parameters, the nature of which are algorithm dependent. Conventional parametric surfaces such as *quadrics*, *Bezier*, *B-Spline* [63] and *NURBS* define a parameter space over which sampling can occur. Such surfaces are also differentiable if required. Algorithms based on procedural generation can take any form of parameter, such as local context and scale information. A local point set representing a surface may be refined to some specific resolution in object or image space when required.

Procedural methods might smoothly interpolate surfaces, synthesize 3D texture or design complete environments such as interiors, cities or landscapes. Procedural methods are ideal in that they embody a form of compression through the use of a minimal set of parameters for a function, where potentially infinite detail is available on demand. Procedural methods may be practical if sampling is inexpensive. Procedural detail can also be reproducible if required, even if randomized, by using the same input data, including

seed values. Distributed noise functions are particularly useful in that they associate randomized values with spatial components. Procedural surface enhancement is related to surface reconstruction from point sets such as Hoppe and colleagues [104], Fleishman and colleagues [62], Adamson and Alexa [3], Amenta and Kil [8], Reuter and colleagues [169] and Guennebaud, Barthe and Paulin [82]. Point based LOD algorithms could also be classified as procedural systems including Wu and Kobbelt [219], Wu, Zhang and Kobbelt [220] and Pauly, Gross and Kobbelt [154], as could computational solid geometry (CSG) approaches such as Adams and Dutré [1], Pauly, Keiser, Kobbelt and Gross [156] and Wicke, Teschner and Gross [213].

Various problems exist in integrating procedurally generated scene components. Conflicts can occur in hierarchical data structures, where the addition of increased detail at the bottom of the hierarchy may either require look-ahead down the hierarchy based on knowledge of the domain, or re-calculation back up the hierarchy to the root, to ensure that approximating attributes higher in the hierarchy are conservative. Examples of contentious changes may include point position, size, surface attributes such as colour, normals or normal distributions. The remainder of this section will look at a select number of example procedural systems that are applicable.

Stamminger and Drettakis [194] define a point based framework for procedural generation of scenes using their $\sqrt{5}$ refinement scheme to locally refine grids based on a rotated grid alignment for LOD. Selective refinement is controlled by an *under sampling factor* formulated for displacement mapping, based on image space density. Terrains are also considered as a special case, with occlusion culling. Samples can hierarchically spawn new samples. Procedural *generators* are used to create scene geometry on demand. The system attempts to avoid holes using a technique based on mean sample distances, but this is not guaranteed. Caching is used to exploit frame coherence. Rendering frame rates of about 13 fps are achieved in one example, with procedural modifications at rates less than 10 fps, rendered with 75k points with a 400x400 image size.

An unusual Newtonian dynamics based refinement system is given by Szeliski and Tonnesen [198] with long range attraction and short range repulsion forces. The system is solved using a simple Euler solver. Surfaces can be interactively split, joined or

extended. Surfaces can stretch or grow based on the insertion of new points to maintain the surface sampling distribution, without knowledge of continuity or direct connectivity. Sparse data can be smoothly interpolated. Rather than render splats, the surface is triangulated using a 3D Delaunay triangulation.

Fleishman and Cohen-Or's Progressive Point Set Surface [62] also refines by adding filtered offset data to parametric surfaces hierarchically based on a moving least squares (MLS) approach, introduced by Levin [128].

A *point octree* is used by Pajarola [151] that is adaptively adjusted to sample the dataset, rather than spatially regular, using a split point in each octant that is the mean of the sample points. An object space point interpolation scheme is used to increase sampling density, using a weighted blend of control points. Visibility splatting is used to establish visible surfaces, before a blended splatting pass using GPU based vertex and fragment programs. Frame rates vary from 2fps for the David model, to over 10fps, depending on level of detail on an Intel Pentium 4 2.8GHz system with nVidia GeForce FX5900.

Guennebaud, Barthe and Paulin [83] define a framework for realtime point cloud refinement using interpolation based on a local neighbourhood with support for surface attributes. Their method is fast, but does not guarantee G^1 continuity. Local regions can be selectively refined. Local neighbourhoods are found using a 3D grid system. A neighbourhood operator defines a set $\Psi(p)$ of point subsets $\Psi_i(p)$ about point p, forming a polygon fan. An interpolation function inserts one point for each $\Psi_i(p)$, based on points in $\Psi_i(p)$. Each new point added at the center of gravity in $\Psi_i(p)$, is smoothed based on the local geometrical case. Bezier curves or patches are used when possible, or a geometric alternative is used when local geometry does not support Bezier functions. EWA splatting is used for rendering. Very fast refinement rates of about 1M points per second are achieved using an AMD 2GHz Athlon system with nVidia GeForce FX5900 GPU.

Guennebaud, Barthe and Paulin [82] also define an interpolation framework for smooth manifold point set surfaces using a *one-ring* neighbourhood centered about refined points. Interpolation uses a *point normal* triangle based on a bezier triangle. Large holes in surfaces can be smoothly filled in object space. Performance in this system refines

about 700k points per second on an AMD Athlon 3500+ with an nVidia GeForce 6800 GPU.

Reuter and colleagues [169] selectively refine surfaces with global and local reconstruction on a set of unorganized points using radial basis functions. Their method is based on the defining an implicit surface through the minimization of a *bending energy*. Surface attributes are also reconstructed. Local reconstruction is used when the point density is high. A linear system is solved to find weight values. Large matrices become sparse due to compact local support and are more quickly solved using an iterative solver. Their system also allows for the fast insertion or deletion of points. Although this method is slower than examples such as [82] or [83], it has a specifiable surface continuity capability. A model with 2,832 points is shown to be reconstructed in a range from 1-20 seconds, depending on reconstruction type, with rendering times of between 3-13 seconds for a 256x256 window using an Intel 1.7GHz Pentium 4 CPU.

2.9.4.6 Hybrid Methods

Hybrid rendering systems generally combine two or more techniques that are often perceived as established methods on their own. In the case of PBR, hybrids are generally PBR algorithms with some form of polygonal rendering backup.

This highlights one of the practical problems of using PBR algorithms with current scenes and hardware. Environments that contain large planar polygons are generally not designed for realistic appearance, particularly at close quarters, unless they happen to represent the intended surfaces accurately. This is more commonly the case in man made environments. A cut off point will exist, which is dependent on each PBR algorithm in question, where rendering a polygon with a given image area in pixels will be more efficient than rendering the same area using discrete point samples. Many current scenes have near or sub-pixel polygon sizes under typical projections. Future algorithms and scenes are likely to make guarantees of unique pixel level geometry throughout a scene at any viewpoint, offering support for high quality, high bandwidth images. Until this becomes practically possible, hybrid methods are a viable solution for some scene types.

A very specific scene representation format is given by Cohen, Aliaga and Zhang [37]. The system is a hybrid renderer that replaces polygons with hierarchical, multi-resolution point based representations at runtime, using data resulting from pre-processing of the polygonal scene. It is in some ways similar to point grouping trees used by many algorithms including polygon systems, such as Hoppe [100] [102] or Schmalsteig and Schaufler [181]. Instead of using a tree of implied point grouping operations, a directed graph of arbitrary operations is represented explicitly, called a Multi-Resolution Graph (MRG). Operations at graph nodes can be either a triangle simplification, a point replacement or a point simplification. Unique nodes are identified at the top and bottom of the graph which represent the object in its lowest and highest resolutions respectively. The state of a scene or object is specified by the set of active most recently executed nodes in the graph. At each object representation, local regions have multiple resolution alteration options which are chosen based on a selective refinement scheme.

POP by Chen and Nguyen [29] is a hybrid based on QSplat by Rusinkiewicz and Levoy [171] [172], with added visibility splatting. Triangles are placed at the leaves of the hierarchy, with points at the interior branching nodes. The primitive type is based on the image space size of the node to be rendered. Frame rates of several frames per second are achieved for objects with about 320k points on an Intel 800MHz Pentium 3 system with an nVidia GeForce 2GTS graphics card. The authors claim speedups and improved image quality. The examples given use small polygons that cover a low number of pixels. Scenes with larger flat surfaces may yield greater speedups.

The Randomized Z-Buffer algorithm by Wand and colleagues [205] is an example of a point and polygon based hybrid. This method samples polygon geometry at runtime, selecting whether to pass polygons to a PBR or conventional polygonal rendering pipeline (see Section 2.9.4.2).

Far Voxels by Gobbetti and Marton [71] uses a hybrid voxel and polygon system using splatting and Guthe et al. [88] also combine points and polygons. Both systems also incorporate out-of-core rendering.

The Sequential Point Tree by Dachsbacher, Vogelgsang and Stamminger [48] is a hybrid, GPU centric architecture that flattens the output of a point based hierarchy, to

reduce primitive bandwidth from CPU to GPU and render mostly on the GPU, reducing CPU load substantially. The authors use disc surface elements stored in an octree. Selective refinement is based on an image space error metric. Flatter regions are represented by larger discs. A usable distance range is specified for each node in the hierarchy, with no ranges down the hierarchy overlapping. A sequential list is generated, based on decreasing start distance. Notably, view volume culling is sacrificed for on GPU processing. The performance is notably high, rendering about 77M points per second using an Intel 2GHz Pentium and ATI Radeon 9700. Rendered points are about 55M per second with back face culling. A complex scene consisting of several complex objects and trees is rendered using points, with ground and sky rendered using polygons. High frame rates of between 36-90 fps are achieved with only 5-15% CPU load.

Kalaiah and Varshney [113] use the *differential point* representation to always render surface regions using polygons rather than points, though the original surface samplings are based on point samples with an associated area. This method is not actually a hybrid in a strong sense as the DP is a more advanced form of splatting, a process which often involves polygons in other PBR systems.

Mantler and Fuhrmann [141] import high resolution microscopy data into a grid format at 2000x2000 resolution with colour. The grid is split into marginally overlapping chunks for separate processing and a hierarchy of lower resolutions constructed, for LOD. Holes are prohibited by the uniform sampling process. Polygons are used instead of points when image region coverage is under sampled and sparse. The authors notably do not achieve a speedup over a standard polygon rendering alternative.

Coconu and Hege [36] store both points and polygons in an octree, with one octree per object. Points in each octant are stored in a layered depth image (LDI) type data structure. Triangles are used if the point density is too low. A back to front rendering order is used to support transparent surfaces, in place of visibility splatting (see Section 2.9.5.5). Point sprites are used with texture coordinate processing to make splats appear elliptical. Using an Intel 2GHz Pentium 4 with ATI Radeon 8500 graphics card, performance for a 500x400 image achieves about 1.3 fps for 2 pixel splat size, rendering a large scene with distribution of 200x200 instanced models conventionally comprising of 10^{10} polygons.

2.9.5 Image Rasterization

Point based rendering algorithms use their respective data structures and rendering processes to result in a set of point samples that must be used to draw the image. If occlusion culling is used, ideally, only image samples that are visible in the first layer will be included, potentially with some conservative over estimate, resulting in fewer over all samples. Techniques that do not use occlusion culling may have greater numbers of point samples and require more rasterization and visibility order determination.

This section looks at some of the issues and solutions regarding the generation of a final image from various PBR algorithms.

2.9.5.1 Overview of Basic Splatting

Splatting in general is the process of projecting an object space primitive in some fashion such that it has an image space representation that sufficiently represents its shape and image space area, or *footprint.* Splatting has been used for some time in the volume rendering field, introduced by Westover [210] and has been widely employed in PBR systems due to its suitability for projecting small, discrete, uniformly specified samples with speed and simplicity, relative to perspective correct polygon rasterization. An early hierarchical method is the octree splatting by Laur and Hanrahan [127]. The volume rendering field is particularly interested in approximating opacity integrals through layers of semi-transparent media in a volume using traditional rendering hardware. Typically, point based rendering is more concerned with opaque boundary representations.

Hardware splatting rasterization approaches generally use point primitives or point sprites. The advent of the GPU has brought greater versatility to basic point rendering primitives, whereby their appearance can be greatly altered by vertex and fragment programs for greater accuracy in shape, blending or attribute modulation (see Section 2.9.5.7).

Although splatting is commonly used, it is simply an image space solution to the problem of under sampled surfaces or images, that interpolates samples to stand in for missing detail. Pre-filtering however, such as in the EWA filter (see Section 2.9.5.6) also band limits output to reduce aliasing. As hardware speeds, memory and software tech-

nology improves, it would appear likely that sufficient scene detail can be provided and rendered, such that all pixels receive unique geometry in all views, alleviating the need for splatting, although band limiting may still be required for anti-aliasing.

The 2D splat representation is often an approximation. For example, the shape, area or depth may not represent that of the object space primitive precisely, in an attempt to maintain high frame rates and may not have perspective correction. Often, it is necessary to restrict algorithms to image space primitives such as OpenGL points, or point sprites which are quickly rendered or composited by existing hardware. More recently, fast fragment shaders in GPU architectures [60] allow for more advanced image space splat evaluation, to affect shape or more accurately represent surface attributes.

Point based surface representations that form contiguous surfaces without geometric holes, such as QSplat [171] [172] can render simple splats using opaque primitives. More complex rasterization approaches such as surfels [158] [168] reconstruct surfaces from overlapping samples using kernels (see Section 2.9.5.6). If sampling may not result in contiguous surfaces, holes must be located and filled (see Section 2.9.5.4).

2.9.5.2 Contiguous Surface Splatting

Polygonal objects represent surfaces using contiguous surface representations, as do implicit, parametric or voxelized surfaces. This property means that those surfaces are also contiguous under perspective projection in image space. PBR methods can also represent contiguous surfaces in object space, potentially providing the same contiguous guarantees in image space if elements are accurately projected or are conservative in their footprint. The most explicit example is QSplat [171] [172]. This method conservatively approximates a local mesh region around an object vertex using a sphere which is a guaranteed super set of the local mesh's area (see Section 2.9.4.1).

This method is hierarchical in that each parent sphere volume is guaranteed to contain its child volumes. As any approximation at any level of this hierarchy is a conservative super set of the area coverage of the polygons associated with its leaf samples, any particular combination of varied local resolutions across a surface can be chosen from the hierarchy, whilst guaranteeing a contiguous surface.

QSplat uses circular, square or elliptical splats to approximate the appearance of object space spheres. Elliptical splats do not have conservative pixel coverage in image space with respect to an accurate projection of their object space spherical representation, potentially resulting in holes. Kernels can also be used to modulate the opacity of the splat. The authors use a spline to specify a fall-off which reaches $\alpha = 1/2$ at the splat radius specified by their projection.

Another example of contiguous splatting is the polygon sampling method by Cohen, Aliaga and Zhang [37] which tessellates overlapping discs to guarantee coverage of a polygon in object space.

2.9.5.3 Surface and Image Reconstruction

In contrast to contiguous surface representations that can be splatted simply, other approaches sample surfaces in more complex ways, also in the knowledge that some regions of the image could be sub-sampled, causing holes which need to be filled.

Grossman and Dally [79] [80] sample objects using a distribution based on a mesh of equilateral triangles in an orthographic projection, to a specified density, accounting for oblique surfaces. Points are projected, but holes may exist. A method termed *pull-push* is used to detect and fill, based on a hierarchical z-buffer approach (see Section 2.9.5.4).

The Surfels method by Pfister and colleagues [158] uses a method termed *visibility splatting* (Section 2.9.5.5) to firstly establish the set of visible splats to permit local surface reconstruction and secondly, to detect holes (see Section 2.9.5.4). Their surfels are tangent discs on the surface of the model whose radii are specified by the longest distance to a neighbouring sample. Under substantial curvature, these discs will not seamlessly tessellate the model's surface, leaving gaps. Layered depth cubes in their octree sampled object representation are rendered with LOD control using hierarchical block selection based on estimated samples received per pixel. A speedup is used to project, known as *incremental block warping* developed by Grossman in his point rendering M.Sc. thesis [79]. The tangent disc shape is approximated using an orthographic projection to image space ellipses approximated by squares or parallelograms. An Elliptical Weighted Average (EWA) filter can be used for surface reconstruction (see Section 2.9.5.6).

The Edge-Point Image based system by Bala, Walter and Greenberg [11] is unique in that it samples viewpoint dependent scene silhouettes and interiors with hidden surface removal and reconstructs the surfaces and image from this sparse sampling using a look-up table approach (see Section 2.9.4.4).

2.9.5.4 Hole Filling

Methods that do not initially guarantee contiguous sampling of surfaces in image space, must locate and fill holes to complete images. Two methods will be briefly discussed here, that of the multi-resolution z-buffer method by Grossman and Dally [80] and hole filling in the Surfels system [158] based on visibility splatting and local image sampling.

Filling holes due to insufficient image sampling, in its basic form, is a classical scattered data interpolation problem. A high quality solution would be to fit a C^2 second derivative continuous surface to the attribute components of the image space samples. Unfortunately, accurate classical solutions such as spline surface fitting are far too slow to reconstruct images at interactive frame rates.

The method used by Grossman and Dally [80] is based on the *pull-push* method by Gortler and colleagues [74]. A hierarchical z-buffer similar to that of Greene, Kass and Miller [78] is used to detect and fill holes in the image sampling. The highest resolution in the pyramid matches that of the output image. Successively lower resolutions approximate four pixels in the higher resolution. Image samples are rendered to the z-buffer hierarchy to a level where no holes are present at some level k of the hierarchy. This level has at least one image sample per z pixel. Weights are assigned to pixels such that a value of 1 is trust that the pixel represents a foreground surface sample and 0 denotes a hole pixel. A first stage calculates these weights for all pixels using the z-buffer hierarchy. For a given level in the hierarchy, the pixels in the depth image are considered to be a mesh of squares with vertices at the centres of the pixels. Each of these squares will cover 4^k pixels in the original image at $k = 0$. For a level k, the depth of pixels at $k = 0$ are compared with those at the corners of square in k which contains them. Each corner of the square is assigned a *coverage* value of 1 if it is in front of the image pixel, or 0 if it is behind. The coverage of the image pixel by the lower resolution z-buffer k is then cal-

culated as a bi-linear interpolation of the four corners at the image pixel position. This has weighted the original image space pixel based on its coverage at z-buffer pyramid level k. This process is repeated for all k up to the source image level $k = 0$, adding coverage values for each pixel and clipping the value at a maximum of 1. The pixel weighting is then calculated as $weight = 1 - coverage$, as pixels which are substantially covered in the muti-resolution z-buffer surfaces by those of their neighbours are more likely to be holes. Now that pixels are weighted, those with $weight < 1$ must be filled by calculating a colour value for them, thus filling the holes. To do this, a hierarchical image pyramid is calculated, averaging groups of 2x2 lower resolution pixels. This is a common task, equivalent to a *haar* basis wavelet filter or a mipmap used to anti-alias textures in conventional rendering pipelines. The weight is also calculated hierarchically, as the sum of those in the lower resolution again clipped to 1. In Gortler's method [74], this is the *pull* phase. The *push* phase re-calculates colours of pixels with $weight < 1$ using the next lower resolution's pixel colours. The interpolation is performed using the hole pixel's own weight and colour and the colours of the three closest pixels in the lower resolution image with weights $1/2$, $1/4$ and $1/4$ based on proximity.

The Surfels system by Pfister and colleagues [158] uses visibility splatting (see Section 2.9.5.5) to create a z-buffer representing visible surfaces to support local surface blending functions, discarding obscured surfaces further away. Pixels which are not filled are marked as holes. When blocks of surfels are projected, an estimate is made of the maximum distance between neighbouring surfels of the block in image space. This is used as the minimum radius of pixel filters, centered at holes using a radially symmetric Gauss filter. Image quality appears to be very high. Additionally, the authors have also implemented a *pull-push* algorithm similar to that used by Grossman and Dally [80] and a supersampling method. The surface reconstruction and pre-filtering uses the EWA filter system (see Section 2.9.5.6).

2.9.5.5 Visibility Splatting

The *visibility splatting* technique applied to point based rendering by Pfister and colleagues [158] permits the incremental accumulation of results in a target buffer, discarding the involvement of obscured surfaces. Visibility splatting can also be used to detect

holes in surface projections (see Section 2.9.5.4) for subsequent filling for non contiguous surface or image sampling schemes.

The visibility pass renders all splat geometry to the z-buffer, but not the image buffer. In addition, a pointer is stored for each pixel, recording the nearest surfel. This creates a depth image from the viewpoint, used to disclude hidden surfaces during subsequent passes. Rendering is then carried out using a depth test, but not a depth write. A small positive depth offset usually applied to the first pass depth writes, or subsequent pass splat fragment depths can add a negative offset, to enable evaluated fragments to pass the depth test for visibility and accumulation in the target buffer. Many other point based rendering systems use visibility splatting as a first pass, with subsequent passes to render visible surfaces. Examples include Xu and Chen's Activepoints [222], Pajarola's Confetti [151], Guennebaud and Paulin [84], Guennebaud, Barthe and Paulin [81], Botsch and colleagues [20], Xu and colleagues [223], Zwicker and colleagues [233], Talton, Carr and Hart [199], the POP system by Chen and Nguyen [29] and Ren and colleagues [168].

2.9.5.6 Elliptical Weighted Average (EWA) Filter

The Elliptical Weighted Average (EWA) filter developed by Heckbert [94] was first applied to point based rendering in the Surfels technique by Pfister and colleagues [158]. A screen space formulation is given in Zwicker et al.'s Surface Splatting [232], where surfaces described by discs define an object space reconstruction filter that is mapped in to image space. Each reconstruction kernel is locally defined in a 2D tangent frame. The process can be formally defined as:

$$g(x) = \sum_k f_k r_k(M_k^{-1}(x)) = \sum_k f_k r'_k(x) \qquad \text{(EQ 6)}$$

where the surface g is evaluated at image space point x based on all surface points $k \in S$ and their respective reconstruction kernel r. The matrix M maps between the local tangent space and image space. Here, the inverse of M maps the image point back into tangent space to sample the kernel. The desired surface attribute f is then taken as a product. This can then be reformulated as an image space reconstruction kernel r'.

To reduce arbitrarily high surface frequencies in image space, a low pass pre-filter is also applied in a convolution for image space anisotropic anti-aliasing to band limit based on the nyquist frequency:

$$g'(x) = \sum_k f_k r'_k(x) \otimes h(x) = \sum_k f_k \rho_k \qquad \text{(EQ 7)}$$

where the combined function g' includes image space pre-filter h, combined to the unified EWA re-sampling kernel ρ. Normalization of the results are then required by dividing by the sum of the kernel weights. Gaussian functions are used for both reconstruction and pre-filtering. In practice, due to local support of the reconstruction kernels, projective methods can be used to evaluate the function. Due to the requirement for accumulation during evaluation, these systems are frequently implemented using the *visibility splatting* technique (see Section 2.9.5.5) to limit blending to local surface regions. An object space EWA formulation is also given by Ren, Pfister and Zwicker [168] that is better suited to GPU processing.

2.9.5.7 Splatting Enhancements

Splatting for point based rendering, often based on the *surfels* system by Pfister and colleagues [158] has been widely applied in the literature. This section will briefly examine some enhancements to increase the features, quality, performance or adapt splatting techniques to GPU hardware.

The EWA filter used by Pfister and colleagues [158] [232] has been adapted to an object space EWA filter by Ren, Pfister and Zwicker [168] for suitability in GPU based splatting, increasing the splat rate by an order of magnitude, achieving frame rates of about 25fps with an object consisting of 100k surfels on a 512x512 display. EWA filtering has also been used in volume rendering by Zwicker and colleagues [231].

Botsch, Spernat and Kobbelt [21] derive linearly varying normal fields over splats by fitting least squares planes to local neighbourhoods to incorporate phong shading with perspective correction and introduce splat clipping for sharp features. Their GPU based system achieves about 4M splats per second on an Intel 3GHz Pentium 4 CPU and nVidia GeForce FX5950.

Botsch and colleagues [20] demonstrate an EWA based splatting system developed on GPUs, using OpenGL points rendered as squares. Three passes are used implemented as GPU fragment programs. The first applies visibility splatting, the second evaluates surface attributes and the third applies deferred shading. Performance of about 20M splats is reported using an NV40 based nVidia 6800 Ultra.

The EWA filter has been extended by Zwicker and colleagues [233] for increased perspective accuracy of the kernels, implemented using the OpenGL point primitive combined with a GPU fragment program. The authors also include a splat clipping process to define sharp edges. This system achieves a throughput of about 2M splats on a platform with Intel 3GHz Pentium 4 and nVidia GeForceFX 5950.

Guennebaud and Paulin [84] use surfel rendering with visibility splatting and extend splat rendering to incorporate per fragment parallax depth correction, based on OpenGL point rendering and a GPU vertex and fragment programs, achieving performance of about 9fps for an object with 400k points on an AMD Athlon 2800+ CPU with nVidia GeForce FX 5800 graphics card.

Talton, Carr and Hart [199] use splatting to derive a Voronoi approximation to a sparse set of samples. A first pass uses visibility splatting. In the second pass, overlapping splat regions are associated with their closest splat using a z-buffer based technique. The distance to the splat center is tested and potentially written to a z-buffer, rather than the depth value. In this way, each pixel can be associated with the nearest point. An additional pass is then used to accumulate each attribute type over the surface. The authors achieve a throughput of about 500k splats per second using an AMD Athlon 64 3500+ with nVidia GeForce 6 series GPU and an image size of 1024x768.

Xu and colleagues [223] have extended the visibility splatting technique to render silhouette boundaries, primarily for artistic purposes. In a first visibility splatting pass, oversized splats are written to the depth buffer. In a second pass, normal sized splats are rasterized to the image buffer. The over sized splat boundaries are not over-written and are used to identify silhouette regions at pixel level.

2.10 Summary

This chapter has covered previous work in scalability and optimization for both polygon and point based rendering. It has covered scalability solutions for locale management, level of detail and control systems, particularly selective refinement systems. It has also covered visibility ordering, view volume, back face, occlusion culling and image based rendering. A range of point based rendering solutions has been examined in terms of scene sampling, scene representation and rendering techniques.

A small number of systems have combined level of detail and occlusion culling optimizations in polygon based systems, including The Berkeley Walkthrough system by Funkhouser and colleagues [65], the MMR system by Aliaga and colleagues [7], Gigawalk by Baxter and colleagues [14], Andújar and colleagues' [9] Hardly Visible Sets (HVS), El-Sana, Sokolovsky and Silva [54], Yoon, Salomon and Manocha [224], Govindaraju et al. [76] and the Far Voxels system by Gobbetti and Marton [71] that uses splatting.

Combination of LOD and occlusion culling in point based systems is rare, but includes the Deferred Splatting system by Guennebaud, Barthe and Paulin [81] and the Layered Point Cloud system by Gobbetti Marton [72]. These occlusion culling approaches are generic, hardware occlusion test based methods that do not specifically leverage properties of the point based representation. A terrain specific occlusion culling technique is also given by Stamminger and Dretakis [194], but is not generally applicable to arbitrary scenes.

Image based rendering techniques have been discussed, with respect to image caching, impostor based approaches [177] [185] that are most readily applicable to the optimization of real-time rendering algorithms. Notably, the introduction of depth [178] and particularly of layered depth images [184], is advancing impostor techniques towards point based rendering, with one explicit example [216].

Most current point based systems are concerned with rendering quality, rather than large scene scalability and therefore tend to use single objects rather than embed the rendering technique in a scene graph architecture.

The large majority of point based systems rely on hierarchical scene representations with viewpoint dependent selective refinement methods that are typically based on image space size, sampling density or error metrics alone and have generally not yet included other viewpoint dependent metrics examined in Section 2.3.5.

Point based scene sampling, randomized sampling, procedural generation and hybrid approaches have also been examined.

The Canopy algorithm described in Chapter 3 onwards, has a number of properties that overcome various problems exhibited by existing solutions discussed in this chapter. Scenes are sampled using a voxelization technique that captures more detail than Rusinkiewicz and Levoy's QSplat [171] [172] vertex patch sampling described in Section 2.9.4.1, whilst also guaranteeing a contiguous surfaces, unlike Pfister and colleague's Surfels [158] or Grossman and Dalley's point based system [80] and thus requires no hole filling, described in Section 2.9.5.4. The voxelization technique is likely to be faster than ray casting methods used in Surfels described in Section 2.9.4.1, whilst being geometrically comparable to their 3-1 reduction scheme.

Simple locale management can be implemented that requires no prior knowledge of the type of domain represented, unlike cell and portal based approaches (see Section 2.2.2) that require connectivity information, whilst a similar subset of the scene's regions will be addressed based on the behaviour of the rendering algorithm due to the inclusion of front to back rendering with occlusion culling.

The algorithm combines locale management, level of detail control, front to back ordering, view volume culling, back face culling, occlusion culling and back to front ordering in a single architecture, unlike the majority of the systems overviewed in this chapter.

One of the most important and unique aspects of this, is that Canopy renders by refining a single, integrated, high level *image graph* data structure that is a little similar to the ISG described in Section 2.7.2. This single data structure actively provides support for achieving hierarchical LOD control, a front to back refinement ordering, hierarchical occlusion culling and a fixed level of detail back to front compositing order for rasterization. Hierarchical view volume culling and hierarchical back face culling are incorpo-

rated. These sub-linear scalability features are more tightly integrated into one solution, sharing processes and data stored in the image graph.

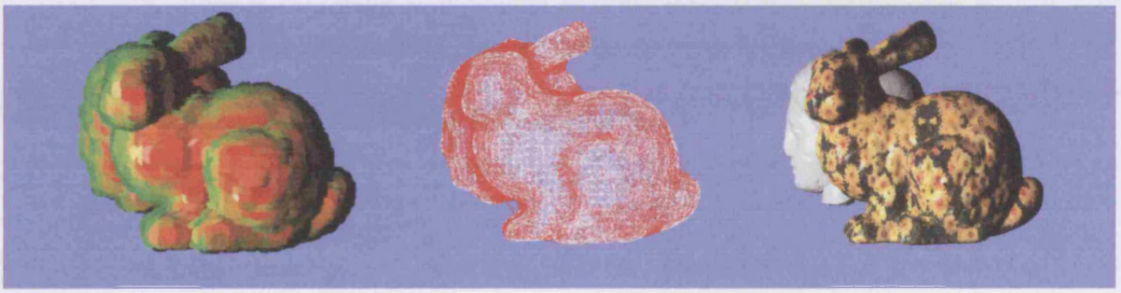
The scene representation offers a homogeneous type that is the same over all scales. The hierarchical scene representation could use many of the vertex grouping schemes for LOD discussed in Section 2.3. Also, it can easily aggregate surfaces, objects and depth layers, unlike many of the LOD approaches discussed in Section 2.3, that are topology preserving. Potentially, less memory is required than exhaustive HLOD approaches. The level of detail control uses a viewpoint dependent selective refinement scheme that whilst simple, can be extended to include more complex control considerations described in Section 2.3.5. Rather than select LODs for specific models that may or may not be visible, like most control systems described in Section 2.3.5, LODs are only refined for objects and surfaces that are likely to be visible in the image, using occlusion culling.

The occlusion culling system is a simple, from-point technique that is more suitable for dynamic scenes than from-region approaches (see Section 2.7.2). The system exploits the geometry of the point based representation, unlike any of those described in Section 2.7, which are typically based on object bounding volume geometry, polygon geometry, or rely on intensive hardware occlusion queries in point based systems. All occluding geometry is culled against, unlike various methods that require pre-selection of a sub-set of the scene as occluders, such as Zhang and colleague's Hierarchical Occlusion Maps [228] (see Section 2.7.2). and does not need occluders to be any specific type of geometry such as a convex model or a large polygon. LOD is not reduced by low occlusion due to the ability to process LOD at surface level. Due to the nature of the homogeneous scene representation, occluders or occludees can be groups of objects, objects or surface regions, in contrast to many systems that simply use fixed level of detail objects. Occluders are also considered in combination to realize occluder fusion (see Section 2.7.2).

The canopy system is also developed within an extensible scene graph architecture that is specifically designed to render very large, highly detailed scenes with standard forms of control for distributing objects. This is less common in point based systems, e.g. QSplat [171] [172] or Surfels [158] that typically only consider the rendering of single objects. The next three chapters describe the Canopy algorithm and implementation issues, with subsequent chapters moving on to look at results and conclusions.

Chapter 3

The 'Canopy' Rendering Algorithm



This chapter introduces a typically sub-linear rendering algorithm dubbed '*Canopy*' (as an analogy between a forest canopy and the *image graph)*, that is used to investigate scalability issues within the framework discussed in Section 1.4. This algorithm uses spheres as a primitive and defines a homogeneous *scene tree* hierarchy to achieve a scale independent representation on which to base rendering and other algorithms. The rendering algorithm is a unified solution with operations for *level of detail control*, *view volume culling*, *back face culling*, *occlusion culling* and *depth ordering*. *Shadows* from point light sources and *collision detection* are also examined to further investigate the algorithm's versatility, in Chapter 4. This chapter examines the rendering algorithm at a high level. Chapter 5 then discusses detailed implementation issues, including how this algorithm can be embedded in a scene graph architecture with compression and read on demand rendering. Further details are given in Bull and Slater [24].

An overview of the ***algorithm's tasks*** is given in Section 3.1, with a ***fast introduction*** to the scene representation and rendering techniques following in Section 3.2. The ***rendering algorithm*** is examined in Section 3.3, followed by details of its underlying ***scene representation*** in Section 3.4. ***Locale management*** is examined in Section 3.5, with ***hierarchical view volume***, ***back face*** and ***occlusion culling*** in Section 3.6 and Section 3.7. ***Rasterization*** issues are covered in Section 3.8, with ***image tree caching*** in Section 3.9. Potential ***pipelining*** and ***parallelization*** are discussed in Section 3.10.

The algorithm's ***complexity*** is looked at in Section 3.11. Lastly a ***summary*** of the chapter can be found in Section 3.12.

3.1 Overview of the Algorithm's Tasks

The scalability issues discussed in Section 1.4 are considered, to design a system that directly addresses each issue. The system breaks down the main scalability problem of a very large and highly detailed scene, into a framework of several sub-problems that are addressed. Each sub-problem can be defined as:

1. Reducing the scene S to a *locale* L that contains all scene components to be addressed
2. Reducing the *locale* L to a subset of the locale V that is in the view volume
3. Depth ordering of surfaces in V to V_O
4. Reducing the view volume objects in V_O to just the front surfaces of polyhedra F
5. Reducing front surfaces F to those visible that are not occluded by others M
6. Limiting the resolution of detail in M to that required to render the image R
7. Terminating the rendering traversal when the image is complete

Solutions to each of these respective sub-tasks are:

1. *locale management*
2. *view volume culling*
3. *depth ordering*
4. *back face culling*
5. *occlusion culling*
6. *level of detail control*
7. *rendering termination*

In addition to the main algorithm, issues of dynamics, shadow determination from point light sources, point-object collision detection and object-object collision detection are also examined (see Chapter 4). Rendering termination is discussed but left as future work.

3.2 Overview of Scene Representation and Rendering

The Canopy algorithm is categorized here as a point based rendering (PBR) technique (see Section 2.9), with small similarities to Clark [35] as early as 1976 and QSplat by Rusinkiewicz and Levoy [171] [172] in 2000. It is one of few algorithms to incorporate LOD and occlusion culling, particularly in a PBR algorithm and to our knowledge, the only non-rasterization based PBR occlusion culling solution. See Section 1.7 for a high level introduction to the algorithm and data structures. Here, we shall review essential points and additional considerations, before examining the algorithm and data structures in this chapter.

To design a PBR system that incorporates the solutions listed in Section 3.1, the requirements for a hole free multi-resolution scene representation and rendering algorithm must be addressed, that achieve the scalability objectives discussed in Section 1.4.

It is important that the system is capable of rendering hole free images. Two alternatives are to ensure a hole free object space representation, as in QSplat [171] or detect and fill holes in the image (see Section 2.9.5). The algorithm takes the former approach to reduce processing during rasterization. The hierarchy of spherical volumes forms contiguous surfaces for hole free rendering, whilst lending additional simplicity of uniformity when viewed from different directions. Spheres are used at all scales, alleviating the need for separate object group, object or rendering primitive representations, whilst naturally introducing a hierarchical multi-resolution representation. The effects of inefficiencies of spherical spatial bounding can be qualitatively countered in level of detail control policies that measure error in image space, potentially to pixel levels of detail.

For algorithm complexity considerations and for practical flexibility, it is useful to perceive the scene data structures as representing an infinitely sized and infinitely detailed scene. This places various constraints on the design of the renderer, such that dependency on scene size or detail is reduced or removed completely. In particular, the rendering algorithm should only be addressing a *working set* of the entire scene that will contribute to the image. This means that additional, relatively macroscopic scene areas or microscopic detail outside the scope of this working set must not be directly considered. In the context of this algorithm, these aspects relate to locale management and level

of detail control respectively. The rendering algorithm never relies on access to the root of the entire scene, or the highest resolution leaf node detail, only that which is local, within the working set.

Scene surfaces are sampled using a voxelization process analogous to a 3D rasterization, to produce intersecting spheres that bound voxels (see Section 3.4.3). Each sphere is represented by a *scene node*. This source set of scene nodes is then used to construct a hierarchy termed a *scene tree* with source samples at the leaves (see Section 3.4.4). This contiguous surface form also guarantees contiguous surfaces at lower levels of detail in the hierarchy because each lower level of detail parent node is a volumetric superset of its higher resolution children. Additional attributes such as surface colour and normal distribution can also be approximated hierarchically.

The scene tree is a binary tree of scene nodes, that offers a larger number of LODs than trees with higher branching factors, such as QSplat [171], at the expense of additional storage and processing. Any technique that calculates a binary hierarchy may be used for construction, leaving opportunities for alternative algorithms with varying performance and quality trade-offs (see Section 2.9.4). Due to the complexity of exhaustive nearest neighbour based grouping solutions, approximations using BSP trees [64], k-D Trees [15], or PCA splitting [98] are preferable. Scene nodes must be permitted to intersect or even contain each other in the hierarchy, so the rendering algorithm must be tolerant of this. A positive side effect is that it relaxes requirements for unique spatial occupancy description that are present in other schemes such as octrees, because separate branches of the hierarchy can represent different data present within the same volume of space. This may lead to options for simpler lazy dynamics schemes where complete scene hierarchy rebuilds are not necessarily required when objects move locally.

If insufficient detail is present for a view of a scene, a splatting approach is used to fill image area (see Section 2.9.5), but by far a preferable option is to procedurally interpolate or create additional detail as required (see Section 2.9.4.5) but is out of the scope of this thesis.

The rendering algorithm performs a number of tasks, listed in Section 3.1, to reduce the working set of the scene tree to be addressed. The first task, locale management, identi-

fies a locale root node, from which the rendering algorithm refines the image solution. Only the locale node's child tree is addressed to render the image. Simple locale management can help quickly cull away regions that are considered to be out of the scope of the local region being rendered. During refinement, higher performance can be achieved if regions can be culled as early as possible. View volume and back face culling can be undertaken at any time. However, occlusion culling queries need information about effective occluders that cover a potential occludee. To enable this query, it is preferable if those occluders are identified before the occludee, to prevent unnecessary processing of scene regions that are later deemed occluded. A front to back traversal ordering identifies such occluders before processing potential occludees and has been used in a number of systems [40] [78] [77] (see Section 2.4). This ordering can then be reversed to achieve back to front ordering for correct occlusion of small conservative overlaps in the set of visible scene regions resulting from the refinement's culling processes, without the use of the standard z-buffer approach.

Level of detail can be exploited during hierarchical refinement, to achieve hierarchical view volume, back face and occlusion culling in addition to deciding when to terminate refinement when sufficient level of detail has been achieved for the required task. However, the algorithm does not focus strongly on level of detail representation efficiency, as it is considered that super-pixel primitive sizes will not be common in future rendering architectures as hardware becomes more powerful. Rendering is thus *output sensitive* because level of detail culling occurs at pixel relative resolutions, making the algorithm independent of the maximum levels of detail present in the scene. The algorithm is thus more focused on achieving specified quality, rather than target frame rates or frame rate consistency, but is open to future extension.

Occlusion culling only initially provides higher performance in a scene with high occlusion, if the cost of identifying occlusion and then culling, is less than the cost of rendering occluded regions. When combined with level of detail control, this challenge is far more severe, because the cost of rendering occluded regions may be far less than in conventional occlusion culling systems that achieve speed-ups by discarding full level of detail geometry.

However, there is an additional benefit to occlusion culling, in that it defines a visible subset of the scene, providing opportunities for caching to exploit temporal coherence during the refinement process and for high resolution rasterization.

A *from-point* occlusion culling technique is used, in preference to other approaches such as *from-region*, due to its suitability for dynamic scenes and viewpoint dependent selective refinement (see Section 2.7 and Section 2.3). This comes at the expense of increased run time calculation over pre-calculated from-region approaches.

The multiple tasks of refining a multi-resolution image solution, hierarchical view volume and back face culling, inducing a front to back ordering, mapping occlusion and culling occluded regions and finally establishing a back to front ordering for rasterization, all share the same underlying data structure and refinement process, to realize a tightly integrated design.

An *image graph* represents the image solution undergoing refinement (see Section 1.7). Each *image node* in the image graph represents the use of a corresponding scene node in the scene tree. Arcs termed *image relations* in the graph record object and image space overlap that is used to enforce front to back ordering and measure occlusion of an occludee by one or more occluders. The graph itself undergoes refinement when a node is refined to its two child nodes.

Rendering progresses away from the viewpoint, establishing occluding surfaces at a specified image space LOD, whilst testing multi-resolution occludee candidates on refinement. In this way, the algorithm uses non hierarchical occluders, but hierarchical occludees, in contrast to the HOM [228] and Hierarchical Z-Buffer [78] that also incorporate hierarchical occluder schemes. The image graph is capable of assessing occlusion of scene regions without regard to discrete objects with occluder fusion (see Section 2.7).

Culling restricts refinement to visible regions and thus defines a hierarchical subset of the scene that underwent refinement. Additional data structures can cache aspects of the refinement that are traversed for data reuse between frames and can also serve as a multi-resolution map of illuminated regions for hierarchical shadow mapping, when the rendering process is used from the perspective of a point light source (see Chapter 4).

Refinement of the image graph to full rasterization levels of detail is currently prohibitively expensive on existing CPU hardware. Therefore, rendering is separated into several stages that operate between levels of detail specified using image based metrics (see Section 1.7). The first stage refines the image graph quickly, without occlusion culling, in the expectation that no occlusion culling should occur at this image scale to minimize large errors. The second stage refines the image graph further, to a medium level of detail with occlusion tests and culling, specifying a finite set of potentially visible nodes. The third stage then refines these potentially visible nodes in the image graph to higher rasterization levels of detail, without image graph refinement.

An image space primitive is required to represent the rasterization of an image node in the image buffer. An assumption will be made that refinement can progress to near pixel or pixel sizes, such that an accurate primitive is of less importance. In many cases, it may simply be sufficient to set pixel colours directly. In cases where this is not achieved, a basic splatting technique is employed (see Section 2.9.5).

3.3 The Rendering Algorithm

This section gives a high level overview of the rendering algorithm's function, that is examined in more detail in subsequent sections. More complex subjects such as embedding the scene in a scene graph architecture, geometric compression, scene graph compression and a framework for procedural generation are discussed later in Chapter 5.

3.3.1 Core Data Structures

The *scene tree* is a binary tree of *scene nodes* that represents the scene. Each scene node represents a point in object space. The leaf scene nodes are likely to be sampled from high resolution polygonal model surfaces. Each branching scene node in the scene tree represents all leaf scene nodes in its child scene tree. Each scene node is geometrically spherical and contains information such as position, radius, diffuse colour, surface normal, normal cone and total surface area. We'll see how each of these are used later in this chapter.

An *image node* is an image space equivalent of a scene node that has been projected into image space. An *image graph* is formed using arcs termed *image relations* between these image nodes to store data on the spatial relationship between pairs of image nodes. Each node can have zero or more image relations. Relations are only formed with neighbouring image nodes that overlap in image space. The image graph is similar in nature to directed acyclic graphs used for visibility ordering in cell complexes (see Section 2.4).

An *image tree* is a binary tree of the image nodes in the image graph, that stores the history of one or more rendering refinements, accumulated over successive frames. The binary branching structure of the image tree reflects a traversed subset of the scene tree, where each image node corresponds to its equivalent scene node. This data structure is optional, but serves as a basis for researching basic temporal caching schemes and implementing hierarchical shadow mapping.

3.3.2 Rendering an Image

The algorithm first constructs a scene tree from scene node samples taken from a source object or scene, the most suitable format being high resolution polygon models. These

tasks will be discussed in Section 3.4 and it will be assumed here that the scene tree has been constructed.

Rendering begins at the *locale scene node*, a scene node somewhere in the scene tree that is identified by the locale management system as a suitable starting point for rendering an image from the current viewpoint. The locale scene node's child tree is the only part of the scene tree that is addressed during rendering. We will look at a basic locale management system in Section 3.5, but for now it will be assumed that the locale scene node is provided to the rendering algorithm in each frame and that the node may differ between frames.

The rendering algorithm is based on progressively refining the image, described by the image graph. At any time, the image nodes in the image graph are a representation of the current image solution. Each image node denotes the use of a scene node and describes its projected image space appearance. When an image node is refined, it is removed from the image graph and replaced by two image nodes that represent the child scene nodes. Both of these child image nodes are inserted into the graph, unless discarded by view volume or back face culling and may be subsequently removed by occlusion culling.

Front to back depth ordering and occlusion culling are of prime importance during the refinement. View volume and back face culling are simpler tasks that can be undertaken whenever required during refinement. The front to back refinement ordering and occlusion culling go hand in hand, because to evaluate whether an image node is suitably occluded, there must be knowledge of its occluders. A front to back refinement provides this information in the order in which it is needed, without spending valuable processing time refining scene nodes that are later found to be occluded.

The two problems of refining down the scene tree to include visible image nodes and refining in a front to back order must be solved in a combined traversal solution. If the algorithm successively refines the nodes that are closest to the viewpoint first, there must come a time when refinement of the closest nodes must cease, so that scene nodes further from the viewpoint can be addressed. Level of detail control can be used to identify when an image node is no longer suitable for refinement, to allow nodes further away to be processed.

Effectively, each depth layer in the scene will therefore be refined to a suitable level of detail t, before moving on to the next depth layer away from the viewpoint. If occlusion culling is carried out during this process, all image nodes that achieve refinement to t can be considered as wholly or partially visible, subject to the accuracy of the occlusion evaluation. The nodes that reach t form a totally ordered subset M of image nodes in the image graph that will be termed the *occlusion mask*, against which, all image nodes in depth layers further away are tested for occlusion. This occlusion test can be carried out along with other culling tests when an image node is introduced as a child on refinement.

Scene nodes in the scene tree may intersect or contain each other, either at the terminal level at which scene nodes were sampled to form a contiguous surface, or due to the mixture of scene node resolutions used in the image graph. Therefore, a recursive nearest first algorithm, such as is suitable for octrees [10], is not correct for the scene tree data structure. See Section 3.3.4 for a discussion. A front to back expected ordering is used, where any particular image graph's depth ordering is correct between scene nodes used. To refine, a parent image node can be identified by the image graph as a nearest node not in the occlusion mask, but is refined to nodes that are not necessarily the nearest. These new child image nodes are inserted, taking their place in the image graph with respect to depth ordering, such that they are addressed in turn.

Image relations in the image graph record spatial relationships between pairs of image nodes for depth ordering and occlusion culling. One of a series of states are classified in both image space and object space. Specifically, these geometric scene node relation classifications are *contained*, *intersecting* and *separate*, with two evaluations, one in 2D image space and one in 3D object space. In the case where a pair of image nodes are separate in 2D, no relation is formed between them because we're only interested in image nodes that overlap in image space. Later, we'll see how these classifications facilitate depth ordering and occlusion evaluation. When a parent image node is refined, it is removed from the image graph, along with all of its image relations. New image relations are then established between one or two inserted un-culled child image nodes and those image nodes with which the refined parent node had relationships.

Image graph refinement can terminate in one of two cases. The first is where there are simply no more image nodes that require refinement. This may occur at pixel or near pixel level of detail in image space, or at some lower level of detail, such that the image graph refinement process is used to determine a set M of image nodes considered visible, that are later refined arbitrarily.

The second termination case is where the image is covered by image nodes, such that it can be considered complete because further refinement will not result in any new image node becoming visible. Without this termination, the image graph refinement may still continue after the image is complete, to consider scene nodes representing depth layers that do not contribute to the image. Even though the occlusion culling process may be efficient, particularly due to level of detail reduction over distance from the viewpoint that allows multiple depth layers to be culled as single scene nodes, it is preferable to break the algorithm's dependency on these further depth layers. The number of scene nodes that may still need to be considered could be very large or theoretically infinite, such that the algorithm would never halt even though the image was complete. Rendering termination due to this case has surprisingly been discussed very little in the literature and is discussed but not solved in this thesis. The most commonly discussed occlusion culling methods such as the Hierarchical Z-Buffer by Greene, Kass and Miller [78], the Hierarchical Occlusion Maps by Zhang, and colleagues [228] or Bittner, Havran and Slavik [17] do not address this issue.

The image graph refinement process results in a set of image nodes M that are considered to be potentially visible in the image. If the accuracy of the occlusion culling is good, these nodes will only represent visible or partially visible image nodes in the first depth layer. If the image nodes in M are of sufficient level of detail, they may be used for rasterization themselves. A back to front ordering for rasterization can be obtained to correctly occlude conservative hidden regions and composite transparency, by reversing the front to back ordering achieved in the second rendering stage. Alternatively, if nodes in M are of a lower level of detail, they may be refined further without image graph updates, to increase their detail, though the correctness of the back to front rasterization depth ordering will be reduced. In this case, an approximate ordering may be acceptable,

or a z-buffer may be used. In Section 3.3.7, we will see how multiple stages provide this option to trade off between performance and image quality.

An image node is rasterized using a *splatting* technique introduced by Westover [211] discussed in Section 2.9.5. Ideally, rendered image nodes will be of pixel or sub-pixel size such that the splat footprint is simply reduced to a pixel colour contribution. However, the image space size can not be assumed, so a splatting technique is used. The implementation in this system uses discs to conservatively represent the elliptical conic section of the spherical scene node under perspective projection to an image node in image space.

3.3.3 Image Graph Refinement Example

To demonstrate several issues of image graph refinement a little more, consider the simplified refinement example in Figure 22, with several iterations (a-j). Each iteration shows the refinement in object space and image space, with the corresponding image graph. Corresponding scene nodes and image nodes are assigned the same letter. The node suitable for refinement in the next iteration is outlined. The viewpoint V is shown in object space, with its frustum view volume in a plan view. An example of what would conceivably be visible in each image is also shown, though this is never explicitly created during refinement until the final image solution. The image graph shows image nodes present in the iteration's image solution, with a *dependency counter* r and implied depth order value s as a tuple (r, s). The dependency counter records the number of occluders an image node has, that have still to be refined. In practice, only the dependency counter needs to be stored. The depth ordering occurs naturally through the in-order traversal of the image graph using these dependency counters. In this example, we will disregard occlusion culling and back face culling issues, assuming that all scene nodes are visible and facing the viewpoint. The image locale node A is wholly in the view volume, so no view volume culling need occur during this refinement. This case is representative of a single object, rather than a scene.

Iteration (a) shows a locale image node A that contains the scene to be refined to a given sufficient level of detail t specified by the client algorithm. This locale node is given to the refinement process as a starting node. In this example, it is wholly inside the view

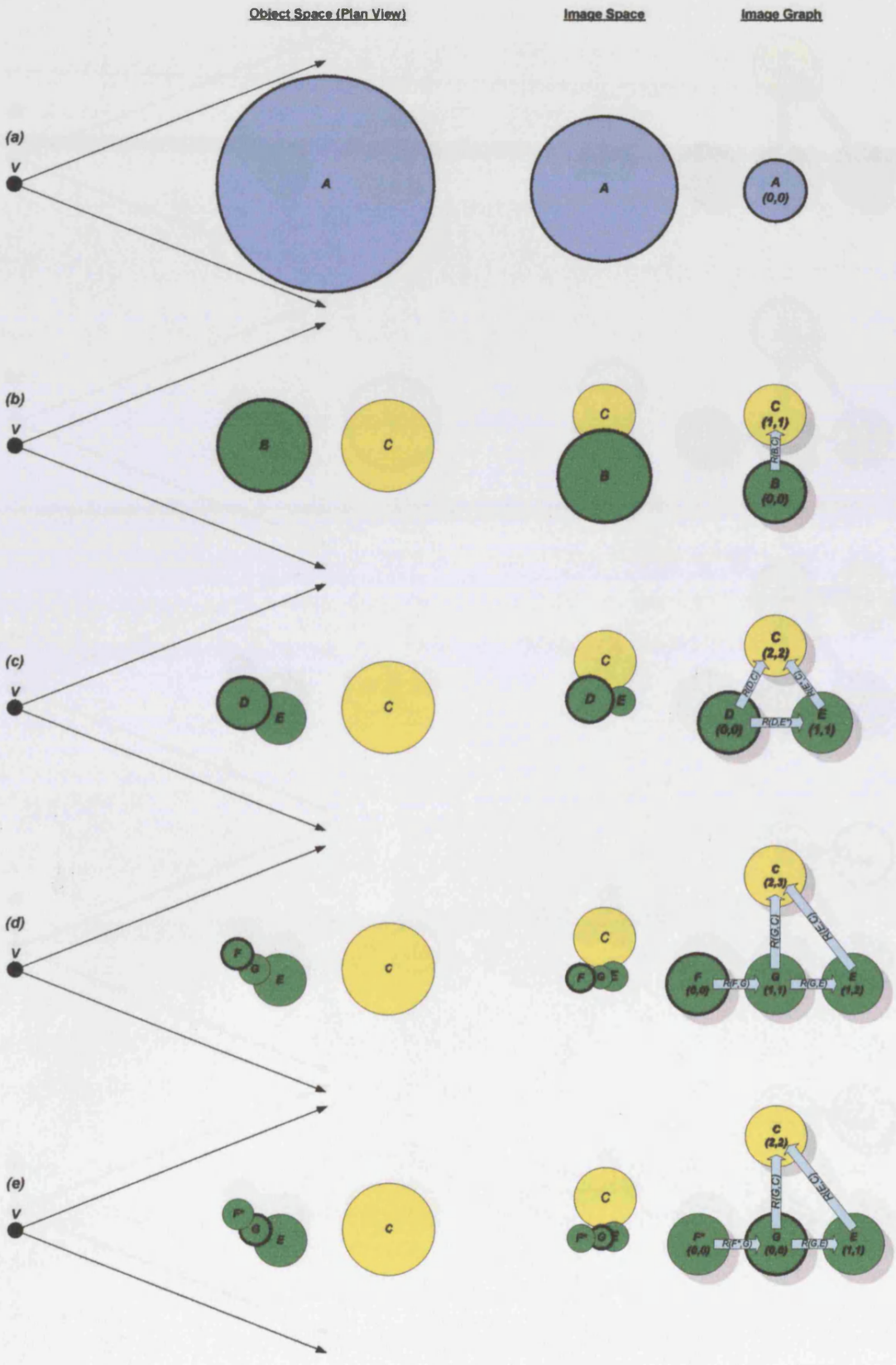
volume and all of its child nodes will have surface normals that face the viewpoint. Node A is due for refinement in the next iteration where $A \notin M$.

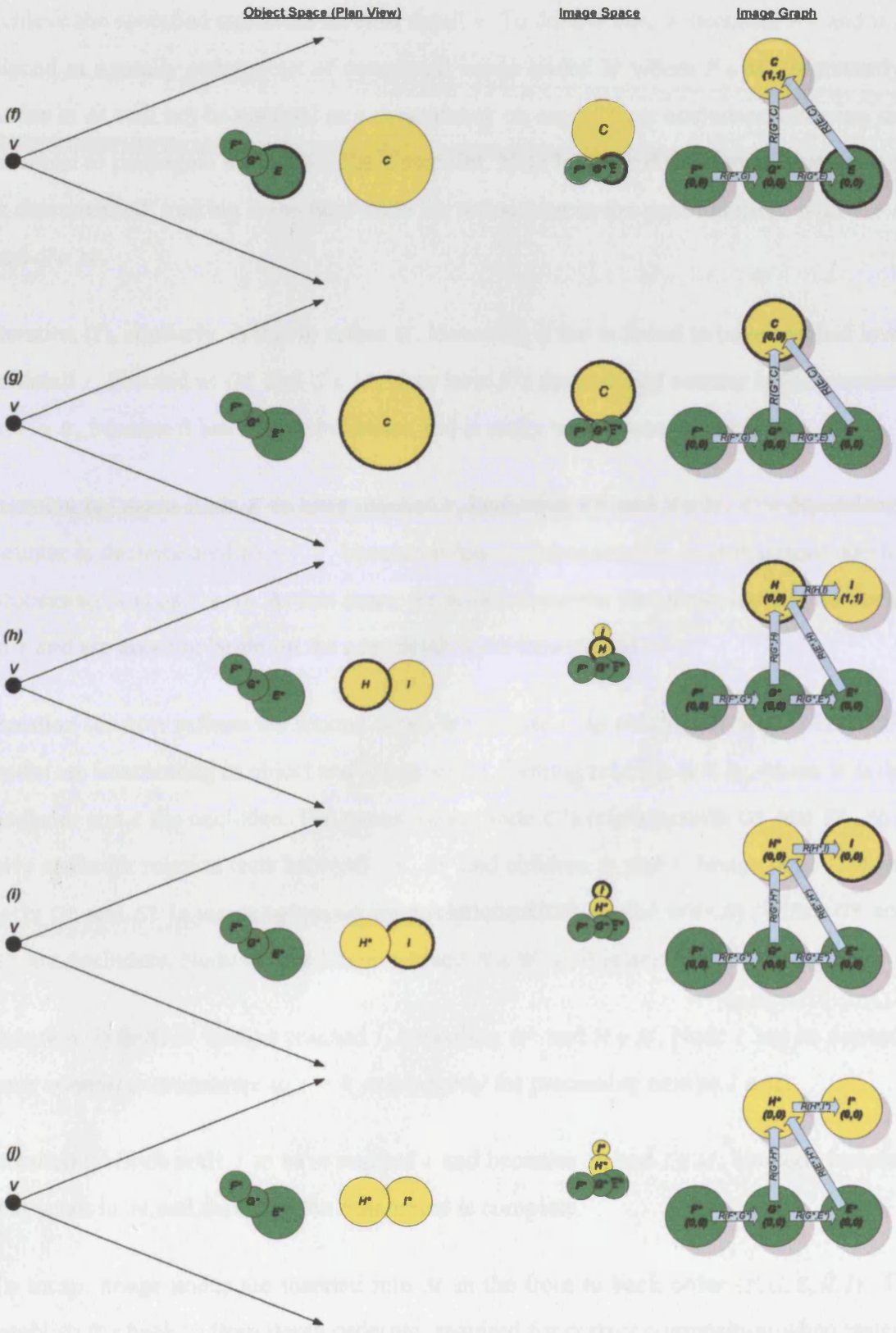
Iteration (b) shows how node A is refined to new child nodes B and C. These two child nodes form two depth layers. Although these child nodes are separate in object space, they are intersecting in image space and therefore have a relationship $R(B, C)$, shown in the image graph. Node B is an occluder of C in image space and therefore B must be processed before C. Image node B has no dependencies and therefore $r = 0$ and depth order $s = 0$. Image node C has one dependency on B and therefore has $r = 1$ and $s = 1$. Note that the image relation is directional, implying that B is dominant over C. If any occlusion evaluation of B or C were required, we could see from the graph that B has no occluders, but C has B as an occluder. Node B is due for refinement in the next iteration because it has no dependencies $r = 0$ and $B \notin M$.

Iteration (c) refines node B to child nodes D and E. These new child nodes are intersecting in object space and image space and therefore have a relation with each other $R(D, E)$. Node D is nearer the viewpoint and is therefore an occluder of E. Their parent node B had a relation with C that must be re-evaluated with child nodes D and E. Because B and C were separate in object space, D and E are also separate in object space from C. In image space though, it can be seen that both D and E intersect with C and therefore relations $R(D, C)$ and $R(E, C)$ are formed because C is the occludee in both cases. Image node D is due for refinement in the next iteration because it has no dependencies where $r = 0$ and $D \notin M$.

Iteration (d) refines node D, to children F and G. The two child nodes intersect in object space and image space and therefore have relation $R(F, G)$ between them, as F is the occluder. The parent node D had a relations with nodes C and E and these nodes are tested with the children for new relations. It can be seen that image node F does not intersect C or E in image space and therefore no relations exist between them with F. Image node G does intersect E in object and image space, forming relation $R(G, E)$. G also intersects C in image space forming $R(G,C)$. Node F is due for refinement in the next iteration because it has $r = 0$ and $F \notin M$.

FIGURE 22. Simple refinement example




Object Space (Plan View)
Image Space
Image Graph
(f)
V
C
(g)
V
C
(h)
V
H
I
(i)
V
H*
I
(j)
V
I*
H*

Iteration (e) is due to refine node F. However, in this iteration, F has been found to achieve the specified sufficient level of detail t. To denote this, F becomes F^* and it is placed in a totally ordered set of completed image nodes M where $F \in M$. Importantly, nodes in M will not be counted as a dependency on any of their occludees, allowing the traversal to propagate away from the viewpoint. Note how the dependency counter in G is decremented, making it the next node for refinement in the next iteration, with $r = 0$ and $G \notin M$.

Iteration (f), similarly, is due to refine G. However, it too is found to have reached level of detail t, denoted as G^* and $G \in M$. Note how E's dependency counter is decremented to $r = 0$, because it has no dependencies and is ready to be processed next as $E \notin M$.

Iteration (g) again finds E to have reached t, becoming E^* and $E \in M$. C's dependency counter is decremented to $r = 0$, because it has no dependencies. C is therefore due for processing next as $C \notin M$. At this stage, we have refined the first depth layer in the scene to t and are about to begin on the next depth layer represented by C.

Iteration (h) now refines the second depth layer node C to children H and I. The child nodes are intersecting in object and image space, forming relation $R(H, I)$, where H is the occluder and I the occludee. The parent image node C's relations with G^* and E^*, now give cause for relation tests between G^*, E^* and children H and I. Image node H intersects G^* and E^* in image space, forming relations $R(G^*, H)$ and $R(E^*, H)$, where G^* and E^* are occluders. Node H now has $r = 0$ and $H \notin M$, so it is next for processing.

Iteration (i) finds H to have reached t, becoming H^* and $H \in M$. Node I has its dependency counter decremented to $r = 0$ and is ready for processing next as $I \notin M$.

Iteration (j) finds node I to have reached t and becomes I^* and $I \in M$. No node remains that is not in M and therefore the refinement is complete.

To recap, image nodes are inserted into M in the front to back order (F, G, E, H, I). To establish the back to front depth ordering, required for correct compositing when raster-

izing the final image, the image nodes in M can simply be processed in reverse order, to obtain the back to front ordering.

This example has shown how the image graph can be refined to a level of detail suitable for a chosen task. The refinement is carried out simultaneously with front to back ordering using dependency counters in each image node. Two separate depth layers are shown in the example, that are treated in order, but it should be noted that ordering also occurs within each layer. The relations formed, track which image nodes occlude which others at their various levels of detail. These relations also serve as the basis for occlusion estimates. Finally, the example also demonstrates how a back to front ordering can be obtained by the process easily, for rasterization.

In this example, there is only ever one image node that is due for refinement because it has a dependency counter $r = 0$ and is not a member of M. In practice, there is a set U of image nodes with $r = 0$ and $U \cap M = \{\}$. Image nodes in U are distributed over the image and are order invariant between themselves. Therefore, image nodes in U can be refined in any order, providing potential for parallel processing.

It is important that the image graph is *acyclic*, such that no dependency loops (transitive closures) can occur that will stall the refinement process. This case will be achieved as long as depth ordering between any two scene nodes can be reliably identified and a deadlock scheme reliably introduces directed sub-graphs between nodes of equal depth (see Section 3.3.4).

If occlusion culling is required during refinement, image nodes in M form the occlusion mask. Nodes newly added to M immediately re-evaluate their occludees for occlusion and potentially cull them. Subsequently, new child image nodes that have relations with one or more nodes in M, undergo occlusion evaluation using those nodes in M to estimate aggregate coverage, that may result the new occludee being culled (see Section 3.7 and Section 5.7).

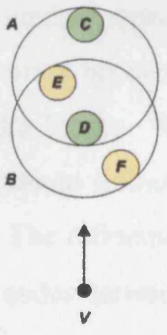
3.3.4 Image Graph Depth Ordering

The example in Section 3.3.3 has shown how depth ordering can be achieved in front to back order during traversal and stored to be used in reversed order for back to front compositing during rasterization. Here, we'll summarize the dependency counting process and look at other more unusual cases that were not present in the example.

Dependency counting is used during image graph refinement to record the number of other image nodes that each image node must wait for, before being considered for refinement itself. An image node's dependency counter must be incremented when a relation with an occluder not in M is added and decremented when a relation with an occluder not in M is removed. If an image node becomes a member of M, all of its occludee image nodes must have their dependency counter decremented to reflect that they are no longer dependent on the node being refined. If any image node's dependency counter reaches zero after an iteration and it is not in the completed set M, it is scheduled for refinement by insertion into set U.

An analogy for the image graph sorting system is a heap of coins. At any time, there is a set of coins that can be removed from the heap without disturbing any others, because no others lie on top of them. Dependency counting is a way of conceptually removing coins from the heap, so that those further down in the heap can be picked.

FIGURE 23. 3D Intersecting parent nodes A and B viewed from V, with D occluding E



Given an image graph, the depth ordering between nodes present in the graph is correct. However, a nearest first refinement policy may not be correct on further refinement of nodes that are intersecting in 3D, because they are mutually occluding. This is why a recursive nearest first refinement policy can not be used. Figure 23 shows an example.

Given nodes A and B, scene node B would be refined first to E and F, because it is nearer the viewpoint. Were the depth first method to then apply the same rule recursively and refine F and then E, the over all ordering would be incorrect when A is refined to C and D because D occludes E.

Image graph refinement therefore uses a probable ordering for intersecting nodes, selecting the closest for refinement first, in this case B followed by A. New child nodes are reinserted back into the image graph sorting system and are traversed in turn. In this case, E will be contained by A and A will refine before E.

A degree of inconsequential out of order refinement can occur with intersecting scene nodes due to variations in LOD. As nodes nearer the viewpoint are refined, those further away remain large in object space. Figure 24 shows separate surfaces S_0 and S_1 viewed from V. At times, nodes may be refined such that they are contained by nodes positioned further away, shown in case (d). The container in a relation that naturally occludes the contained node, is refined first, shown by the refinement of S_1 in (e). Therefore, nodes that are more distantly positioned will sometimes undergo refinement because for a brief period, their surfaces are at the front due to their lower resolution representation. They are refined as required to clarify the front to back ordering.

Further refinement of S_0 in (e) may again cause refinement in S_1. Surfaces in this situation will all gain separate object space classifications in relations between nodes in the surfaces, as they tend towards their boundary representation surfaces. How quickly this occurs depends on the degree of separation between the surfaces in the scene. Surfaces closer to each other are likely to take longer. When discrete, separate depth layers emerge during refinement, their subsequent refinement is guaranteed to be in front to back order with respect to each other. The refinement state shown in Figure 25 has two surfaces S_0 and S_1 where relations in nodes between these surfaces will be classified as *separate* in 3D, if intersecting or contained in 2D. All subsequent refinement is guaranteed to be front to back.

FIGURE 24. 3D Refinement sequence of S_0 invoking early refinement of S_1 in (d)

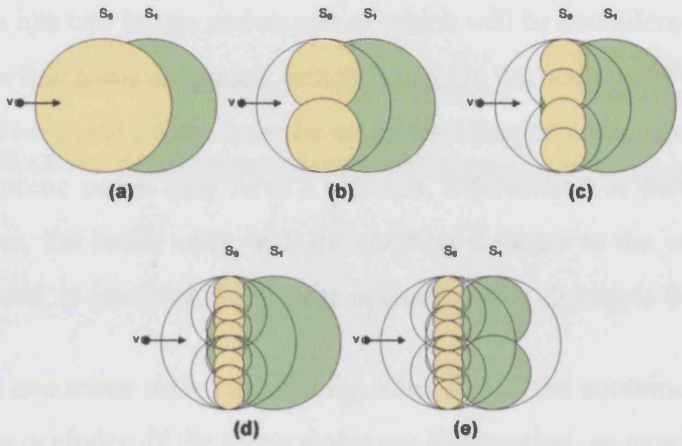
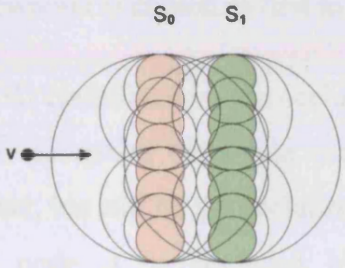


FIGURE 25. 3D Refinement achieving guaranteed ordering through separation



The depth ordering of the final image graph M is correct in itself, between the scene nodes used, at their respective resolutions. Therefore, if the image graph can be refined to high rasterization resolutions, the set M or an image graph traversal could be used to perform a back to front traversal and eradicate the need for a z-buffer to resolve conservative regions left by the occlusion culling and composite transparent surfaces. If the level of detail of nodes in M is larger than a suitable rasterization size in image space, an approximate depth ordering system is possible. Each image node $m \in M$ is refined to a set R_m of image nodes in their child tree without image graph refinement, in arbitrary order to increase their resolution. Each set R_m is then rasterized in the reverse order of the parent image nodes at the lower level of detail in M. The order within R_m must be repeated between frames to prevent temporal changes that may be noticeable.

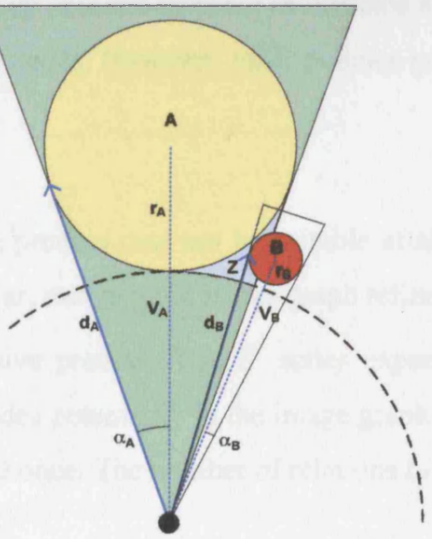
3.3.5 Image Relation Ordering

An image relation has two image nodes, one of which will be considered dominant in the relationship as the first to be processed, which is usually the occluder. For a relation to be formed between two nodes, there must be an intersecting or contained classification in 2D. In 3D, their scene nodes may have a separate, intersection or contained classification. In most cases, the scene node with the shortest distance to the viewpoint from its nearest surface point, is considered to be the occluder, or is chosen to be processed first.

In the case where one scene node contains the other in 3D, the container has the shortest distance and is the occluder. If the scene nodes are intersecting, an exact ordering on further refinement is not determined because of their mutual occlusion and therefore, the nearest of the nodes to the viewpoint is chosen as first to be refined (see Section 3.3.4).

If the nodes have a separate 3D classification, the occluding node in the relationship is usually the one nearest to the viewpoint, but a case exists where it is possible for one node to be further than the other, but still be the occluder, an example of which is shown in Figure 26. The nearer node A is occluded by the further node B, with $|V_A| - r_A < |V_B| - r_B$, where V_A and V_B are A and B's respective view vectors and r_A and r_B their radii. In this example, scene node B clearly intersects A's view cone, forming an intersection 2D relation, is separate in 3D, but is occluding by its presence in area Z.

FIGURE 26. Relation order with far scene node as occluder



Given near node A and far node B with separate 3D, intersecting or contained 2D classification, to test whether B is in A's area Z, we calculate the distance d_B to B's view cone tangental point, as the first point of overlap for B's view cone with A's view cone.

If $d_B < d_A$, where d_A is the distance to A's view cone tangental point that marks the outer limit of Z, then B is in Z and occludes A. For a given node N, the tangental point distance is $d_N = |V_N| \cos \alpha_N$, where V_N and α_N are the view cone vector and semi angle.

3.3.6 Exceptions in Image Graph Refinement

Exceptional cases occur in image graph refinement. A primary case occurs when a scene node contains the viewpoint. It is imperative that these nodes are refined with priority over any scene node that does not contain the viewpoint, so that correct comparisons can be made about discrete scene nodes in the image. If at least one node still contains the viewpoint at high resolutions in object space, the viewer is considered to be against or contained within the volume and the whole image can be set with this colouring.

It is also essential that the image graph is updated with policies that can not lead to cyclic dependencies, where the algorithm would potentially never terminate due to deadlock. These may occur between scene nodes that are intersecting in 3D and equidistant from the viewpoint. To enforce a locally directed graph, a deadlock breaker is used that is consistent between a set of intersecting equidistant nodes. The technique used in our implementation uses lowest memory pointers in place of distance as these are consistent and guaranteed to differ between nodes. However, other policies are also conceivable.

3.3.7 Rendering Stages

The image graph refinement process may not be suitable at all times during the refinement of an image. In particular, carrying out image graph refinement to pixel sized levels of detail may be an expensive process. The 2^n series expansion of the image graph means that the number of nodes potentially in the image graph will double when each of the existing nodes are refined once. The number of relations to be evaluated may be substantial for each image node.

Refinement tasks may differ with image node levels of detail. The locale scene node is likely to be very large and contain the viewpoint. There is little point in attempting to carry out image space occlusion culling tasks at this scale. As the image is refined, there may still be image node scales in image space at which occlusion culling is not suitable, because of the magnitude of the error that would be incurred if an incorrect culling decision was taken due to a inaccurate occlusion evaluation.

For these reasons, the rendering algorithm refines progressively in three sequential stages, before a fourth rasterization stage (see Section 1.7). Each of the three refinement stages carries out a particular set of tasks that are suitable at the image scales to which they are applied.

Stage one is an optional setup stage that refines from the locale scene node, down to a minimum image node size threshold t_1. The image graph is refined in scene tree depth first order for speed. The image graph is not traversed with depth ordering and no occlusion culling decisions are taken at this stage, based on the assumption that no accurate occlusion culling can be made within this scale range and that every image node in the resulting image graph will need further refinement. Hierarchical view volume and back-face culling processes are carried out. This stage results in a correct image graph that is passed to the next stage for refinement.

Stage two refines the image graph in front to back order with occlusion culling. Image nodes are refined to a minimum level of detail t_2 specified as an image space size. Image nodes that achieve level of detail t_2 without being view volume, back face or occlusion culled, are committed to the occlusion mask as members of a potentially visible set M. Occlusion evaluations and possible culling are carried out for new child nodes that have occluding image nodes in M. The resulting image graph and visible set M are then passed on to the next stage.

Stage three is optional and refines image nodes in the given image graph from stage two, also given in depth ordered form in M, to a higher minimum level of detail t_3 suitable for rasterization. This stage is only required if the level of detail t_2 achieved by stage two is too low for rasterization. Each image node $m \in M$ is refined depth first to level of

detail threshold t_3, resulting in a set R_m of image nodes to be rasterized. Stage three permits a smaller t_2 threshold to be used in stage two to limit the amount of image graph refinement that occurs to calculate the potentially visible set M, because updating the image graph in front to back order and calculating occlusion evaluations is potentially expensive. This stage is likely to process the largest number of image nodes of any of the stages. Therefore, to speed up processing, view volume and back face culling may be disabled. This stage results in a set R of rasterizable image nodes, which is the union of all R_m. A totally ordered set of sets $R_m \in R$ is passed on to the next stage for rasterization, where the ordering of R reflects the order in M. If stage three is not required, $R = M$ with only one member per R_m set.

Stage four rasterizes the given set of image nodes R. If the level of detail t_2 in stage two is low, or high quality is required, a z-buffer can be used to resolve depth ordering of image nodes in R. If t_2 is of sufficiently high level of detail, or some level of artifacts in the image are acceptable, each $R_m \in R$ can be rasterized in reverse order reflecting the depth ordering at level of detail t_2, with images nodes in each R_m rasterized in arbitrary, but repeatable order. Artifacts will be least likely when $t_3 - t_2$ is small, t_3 is a high level of detail and each R_m only has one depth layer from a large range of viewpoints. The quality of the results possible are tested in Chapter 6.

The rendering algorithm's structure is shown in the Jackson diagram in Figure 27. The task blocks are broken into sub-task blocks, with the running order progressing left to right. Tasks that are executed iteratively are tagged with *, whilst those that are optional are tagged with O and are mutually exclusive. This variant of the diagram form shows tasks that are called by other tasks, with parameters, shown in the same colour. Note that the system's state is allowed to bleed horizontally and vertically. Parameters used include *locale* root image node L, arbitrary *image node* N *and child image nodes* A, B. If any parameters are NULL, tasks are not carried out. Refinements and culling only operate on image nodes that are not culled.

The main task *Render Image*, is comprised of the four stages described above. A number of sub-task diagrams are also shown including *Refine Image Graph Node Depth First*, *Refine Image Graph Front to Back* and *Refine Image Node Depth First*.

Sub-task *Refine Image Graph Node Depth First* refines the image graph using a depth first ordering recursively, with LOD control, view volume and back face culling, where speed is required and view volume depth ordering is not, in stage one.

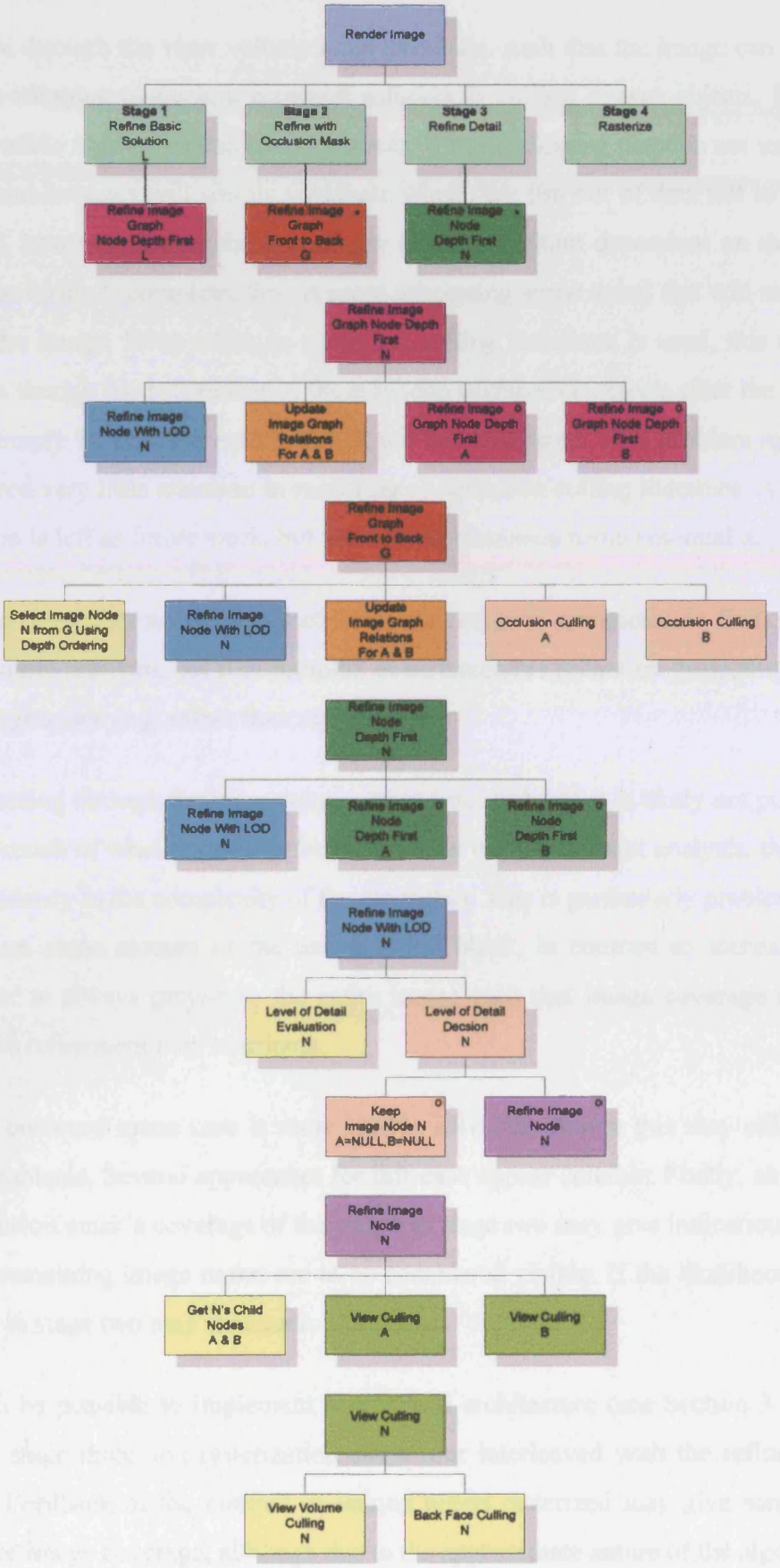
Sub-task *Refine Image Graph Front to Back* refines the image graph with a front to back ordering, with LOD control, view volume, back face and occlusion culling required in stage two.

Sub-task *Refine Image Node Depth First* quickly refines each image node N in G without image graph updates, as required in stage three, with LOD, view volume and back face culling.

Additional sub-tasks *Level of Detail Evaluation* and *Level of Detail Decision* control whether refinement terminates or continues for the current stage. *Occlusion Culling* makes occlusion estimates and culling decisions, whilst *View Culling* carries out view volume and back face culling.

Further details and pseudo code for rendering stages are given in Section 5.4.

FIGURE 27. Rendering algorithm Jackson diagram



3.3.8 Refinement Termination

Propagation through the view volume must terminate, such that the image can be finalized. A far clipping plane is a common solution to culling distant objects, but often results in visible sudden appearance of objects. When a clipping plane is not used, most rendering architectures will simply terminate when they run out of data left to be processed. This, however, makes the complexity of the algorithm dependent on the size of locale. If the image is complete, time is spent processing scene detail that will never contribute to the image. Even when an occlusion culling technique is used, this may still occur, even though the processing of these hidden surfaces or objects after the image is full, may simply be rejected more quickly and not rasterized. This problem appears to have received very little attention in rendering or occlusion culling literature. A termination function is left as future work, but this section discusses some potential approaches.

A rendering algorithm needs to detect when the image is complete. At first, this may appear a simple problem, but it is complex in architectures reliant on projecting geometry into image space (e.g. rather than ray casting).

When traversing through the view volume from front to back, it is likely not possible to know how much of what remains to be traversed is visible, without analysis, thus creating a dependency in the complexity of the algorithm. This is particularly problematic for scenes where some amount of the image is left blank, in contrast to scenes that are known prior to always project to the entire image such that image coverage might be detected and refinement may terminate.

The latter, enclosed scene case is more readily solvable, though this may still present practical problems. Several approaches for this case appear suitable. Firstly, an analysis of the occlusion mask's coverage of the image in stage two may give indications of how likely any remaining image nodes are to be considered visible. If the likelihood is low, refinement in stage two may terminate.

It may also be possible to implement a pipelined architecture (see Section 3.10) with refinement stage three and rasterization stage four interleaved with the refinement in stage two. Feedback of the number of unique pixels rasterized may give some literal indication of image coverage, although due to the approximate nature of the algorithm, it

is possible that some pixels that should be drawn, may not be and this must be taken into account.

An alternative approach may be to monitor the behaviour of the algorithm and terminate when additional visible detail appears to be improbable. For example, an estimation function may analyze the number of nodes in occlusion mask M and the rate at which nodes are being added to M. When this falls off substantially, this may be taken as an indication that no more detail is likely to be visible.

3.4 The Scene Tree Representation

This section discusses the basics of the scene node and scene tree representation. It then discusses how scene nodes can be sampled from existing polygonal models before construction of a scene tree, as input to the rendering algorithm. Later, Section 5.1 to Section 5.3 will cover these aspects in more detail.

3.4.1 The Scene Node

The scene node represents a single spherical scene node. It must encapsulate all of the leaf level scene nodes that it represents and serve as a geometric and attribute approximation. Any additional information that may be required by the rendering algorithm must also be included. Here, we'll examine the attributes *position, radius, surface area, normal cone* and *diffuse colour* and why each is required.

To make the system scalable during any potential run-time reconstruction of the scene tree due to dynamics, a restriction has been placed on scene nodes, such that all information stored within them must only be a function of its two child nodes to prohibit deep downward scene tree analysis that would use substantial resources. Therefore, the scene node attributes must be chosen as hierarchical approximations that serve the requirements of the rendering algorithm, whilst being light weight, fast and scalable to update. Each of the geometric attributes must be also be subjected to affine transformations [63] resulting from scene dynamics.

The *position* and *radius* simply defines where a scene node is in object space and its spherical size. A parent scene node must encapsulate its two child scene nodes.

The *surface area* is the sum total surface area of the regions of the geometric model approximated by the scene node. This is simply the sum of the surface areas of the two child nodes. Surface area must be measured accurately when the leaf scene nodes are sampled. The surface area is used in several ways. Attributes in the scene node can be calculated as a surface area weighted function of those in its child nodes. The algorithm's occlusion culling system also requires an estimate of the strength of an occluder and again, the surface area is used to see how much of a surface is represented by the node.

The surface normal cone is a solid angle centered about a centroid normal vector. The vector specifies a normal for the scene node, whilst the solid angle encapsulates all normal vectors of all scene nodes in its child scene tree. This normal cone is used in several ways by the algorithm. The surface normal is used for diffuse and specular shading calculations, much like any other rendering algorithm [63]. Hierarchical back face culling also needs knowledge of whether a scene node itself, or any scene nodes in its child tree might be facing the viewpoint. If there is no possibility any exist, the scene node is back facing and can be culled. The algorithm's occlusion culling functions also use the normal cone to estimate the strength of a scene node as an occluder, by distributing the scene node's total surface area over the normal cone's solid angle and then measuring how much of the solid angle is visible to the viewpoint.

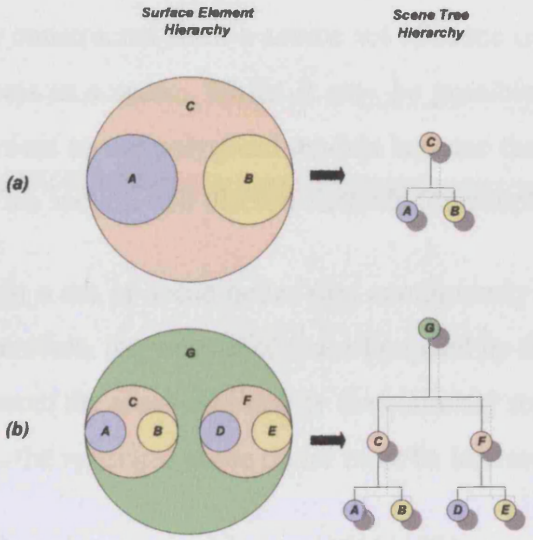
A simple colour attribute is also included to represent diffuse reflectance in the local illumination model. Specular reflectivity is considered to be white. Additional values can be added if required, to give greater shading control. Storing one colour per scene node may appear wasteful when material nodes might be used to specify properties of whole groups of samples. However, texture mapping will not be used in this architecture, so unique colouring is necessary. Moreover, hierarchical surface area weighted colour provides low pass filtering for anti-aliasing. Another benefit of unique sample colouring is that no surface parameterization is required for unique one to one mappings for storing pre-calculated lighting in a process known as *texture baking*. The approach can be extended to include directional colour for more accurate low LODs [71].

3.4.2 The Scene Tree

The scene tree is a binary tree of scene nodes. Algorithms that use the scene tree are independent of the algorithms used to construct the scene tree. Therefore, any grouping or partitioning scheme may be used, without erroneous consequence to the rest of the system. Moreover, it is possible to mix scene tree construction heuristics within the scene tree, for example, if some scene scales may benefit from switching methods.

Figure 28 shows nested spherical scene nodes at two heights (a) and (b), with their associated binary *scene trees*. Given two child scene nodes A and B, a parent C must be constructed that wholly contains the two children.

FIGURE 28. Spherical scene node binary hierarchy at heights (1) and (2)



If the tree is well balanced, with tight hierarchical bounding, more usable LODs will exist at branching nodes and rendering will operate more quickly and efficiently than with a very skewed tree, or a tree with inefficient bounding through bad spatial grouping.

Parts of the scene that are expected to be dynamic, such as individual objects, may be constructed as a distinct scene tree that can be moved as a rigid body, that is integrated into the main scene tree of static objects, without recalculation of the object's tree.

A scene will consist of a very large number of scene nodes. An individual model may consist of many millions of scene nodes. The largest of the scenes demonstrated in Chapter 6 comprises over a trillion scene nodes, by instancing models in the scene. It is therefore advantageous to store the scene nodes and scene tree efficiently using compression methods. Techniques such as scene tree instancing also reduce total storage for replicated objects.

Techniques for compression and embedding the scene tree in a compressed scene graph architecture with instancing are described respectively in Section 5.9 and Section 5.10. Inline file processing and on-demand fetching of data for out of core processing are also described in Section 5.10.7, Section 5.10 and Section A.4.

3.4.3 Scene Sampling

The scene tree must be constructed from a source set of scene nodes S representing an object or multiple objects in a scene. Whilst it may be possible to use point sets as a basis, it is more convenient to use polygonal models because their surface connectivity information is useful. This section will discuss methods for sampling polygon models.

The problem is to obtain a set of scene nodes that contiguously cover the scene's surfaces without holes. Therefore, the volume of space bounded by the set of sampled scene nodes must be a super set of the space outlined by the boundary representation of the polygon model. To do this, the spherical scene nodes must be intersecting.

The QSplat system by Rusinkiewicz and Levoy [171] [172] uses a polygon patch system that places a sample at each vertex in the model, calculating the sphere radius as the longest extent of any polygon that is defined by the vertex (see Section 2.9.4.1). This method whilst resulting in contiguous surface coverage, will result in a set of samples whose size is a function of the surface area of the polygon patch surrounding the vertex. The maximum sampling resolution is also a function of the resolution of the polygon model, which is inflexible. For example, a higher resolution sampling may be used, which also samples bump maps, displacement maps or 3D textures.

A voxelization technique [116] [117] [118] is used, described in detail in Section 5.2 that uses a 3D analogy of a scan line algorithm to create a set of samples on the surface of a polygon model. Each polygon is fragmented into multiple voxel samples. Each cubic voxel is interpreted as a bounding scene node. Attributes such as position, radius, surface area, normal and colour result from the voxelization, using either polygon attributes or vertex attributes that are interpolated across the polygon. Textures are sampled for diffuse colour if defined, with alpha channel for stencils.

Alternatively, other procedural methods may be used to create a set of source image nodes. Note that these image nodes need not be leaf nodes in their own right and may each be assigned a child scene tree. A basic framework for procedural generation embedded in a scene graph architecture is given in Section 5.10.

See Section 5.2 for more details on scene sampling in the Canopy system.

3.4.4 Scene Tree Construction

A scene node hierarchy is constructed from the set of scene nodes S, sourced from a sampled scene or procedural methods. The source set is unstructured and independent of the technique used to construct the hierarchy. Each sampled scene node must include all attributes to represent the surface, detailed in Section 3.4.1.

Assuming a basic definition of level of detail approximation accuracy as a function of bounding volume error and the number of branching nodes (hence tree levels required), the scene tree construction problem can be defined as minimizing the total bounding volume error ε_B and number of tree levels H, where the minim number of levels is $H = Log_2 n$ where n is the number of samples in the source set S.

The choice of hierarchical construction algorithm will have some affect on the accuracy of multi-resolution representations and the functions that rely on them. Tight spherical bounding will give rise to better level of detail approximations.

A BSP tree method with K-D tree like ordering has been chosen for use for its speed of solution and partitioning adaptability at each tree level. Other partitioning methods could also be used such as nearest neighbour based grouping, octrees, principle components analysis (PCA) splitting [98] or mesh optimization orderings, such as an edge collapse sequence [100] that merges vertices in a model (see Section 2.9.4).

The source set of scene nodes is used as the basis for a top down partitioning process. Each sample scene node may be considered a leaf, but can represent higher levels of detail in a child tree that is not addressed by the scene tree construction algorithm, as long as the sub-tree is hierarchically consistent or the whole hierarchy will be updated.

A bottom up grouping algorithm is likely to produce more balanced trees because the total number nodes in each tree level can be carefully controlled by simply making sure that all scene nodes at level $h+1$ are paired to a new parent at level h. However, an efficient method is first required to establish which pairings are most suitable. For speed and simplicity, particularly to support dynamic reconstruction, the system uses top down partitioning, rather than a bottom up grouping approach.

The construction algorithm uses two passes. The first iteratively partitions in a breadth first, top down order, also creating *container* data structures to reduce tree links. This pass also calculates hierarchical attributes in bottom up order. The second top down pass encodes the hierarchy with geometrical compression (see Section 5.3 and Section 5.9.1) and packs scene nodes into container nodes (see Section 5.9.2) to reduce links.

A partitioning operation takes a given set of nodes N at tree level h. Each $n \in N$ is entered into one of two child groups, Q_0 or Q_1 such that $Q_0 \cap Q_1 = \{\}$, based on their scene node position, disregarding their volume. A new scene node is created for each of the two child sets Q_0 and Q_1 that belongs in the next scene tree level down $h+1$. This process begins with the whole scene $N = S$ and is repeated iteratively in a breadth first order, down each scene tree level until the source scene nodes are reached and there are no scene nodes left to process.

Scene node attributes are hierarchically calculated in a breadth first, bottom up order. Attributes such as size, normals, colour and surface area are calculated for each branching node up the scene tree, to the root node.

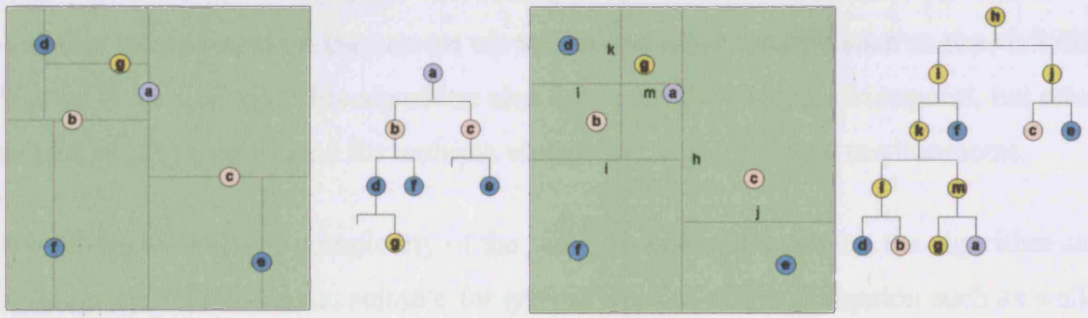
The tree partitioning method chosen is a form of binary space partition (BSP) tree [64] with K-D tree [15] like ordering. A conventional K-D tree sorts using different dimensional criteria at each level, a 2D example of which is shown in Figure 29.

The axis aligned spatial extents E are calculated for each branching scene node. This can either be based purely on the positions of the scene nodes in Q or can only include their radii. A partitioning plane is chosen based on E such that it partitions along the largest extent. This attempts to make each halfspace on each side of the partition compact and as cube like as possible, such that the contained set is compact. According to the typical K-D tree method, axis aligned planes are used, but it is possible to use other non axis aligned planes in a conventional BSP tree form. A non axis aligned selection process is likely to give rise to better results, but the axis aligned method is simple to develop, fast to execute and produces suitable results.

Figure 29(a) shows a conventional K-D tree in 2D, partitioning a set of points (a-g) with partitions chosen at the points themselves. Figure 29(b) shows a variant method, parti-

tioning a set of points (a-g) that remain at the leaves, with new interior branching nodes. This second form in Figure 29(b) is used to provide the hierarchical scene tree representation with multi-resolution approximations of the leaf nodes at branch nodes. Note that all parent nodes have exactly two or zero child nodes. This simplifies the rendering algorithm, compression systems and data structures. Partitioning terminates when the extents of both child nodes is within a tolerance value such that they either contain one or more vertices at the same position or several within a welding tolerance, to merge duplicate or very close proximity scene nodes. The first case where several scene nodes are present at the same position needs to be resolved such that only one is used to represent all information at that point in space. Welding together points that are very close helps reduce the tree depth whilst identifying cases of equivalent points separated by numerical rounding or truncation error. Some degree of error is accepted in this process, where very close nodes either side of a partition will not be welded.

FIGURE 29. K-D Tree partitioning of a set of points (a) Standard, (b) Variant



3.5 Locale Management

Locale management (see Section 2.2) serves as a first level of scene culling before more conventional forms such as view volume culling are carried out during rendering. A subset of the scene is defined that is then applied to the main rendering function. Preferably, rendering will not address any scene data structures outside of the locale. A locale may be chosen through a complex analysis of what the user may consider important in a scene, or simply what is likely to be visible.

Given a scene S with j scene nodes, locale management must efficiently identify a subset $L \subseteq S$ containing k scene nodes of the scene. Ideally, the complexity of the locale management would be independent of the size of the scene j. However, because any scene node in S may be chosen as a suitable locale root node, it is difficult to break this dependency as an upper bound.

Here, we will consider a simple locale management scheme for static scenes that selects a suitable locale based on the current viewpoint and other criteria, such as the visibility distance to the horizon. This algorithm also only considers a single viewpoint, but other variants could be developed for multiple viewpoints, e.g. for shared environments.

To improve the average complexity of the locale management system, the algorithm can exploit temporal coherence, suitable for typical types of scene navigation such as walking or flying. This system will perform worst when coherence is low, for example when rendering from a sequence of randomly chosen viewpoints. Considering L_1 as a given locale node whose child tree specifies the scene subset, the next locale node L_2 can be found using a local scene tree search at some later time, for example in the next frame.

The chosen locale must embody all scene data that will enter the rest of the rendering algorithm. This poses a problem for the scene tree data structure, because scene nodes are naturally permitted to intersect or even contain each other to facilitate the representation of contiguous surfaces and relaxed, non optimal hierarchies for faster dynamics. The scene node chosen as a locale would not necessarily embody all scene data that exists within its volume. One solution may be to only select scene nodes that are identified during the scene tree's construction as having no intersections or containment with scene

nodes outside of their own child scene tree. However, these cases may be rare and provide few locale candidates. Another alternative is to store object space connectivity information in scene nodes, such that a node that is selected can have a temporary scene tree constructed above it to unify intersecting and contained nodes that must be included, or traverse to a higher scene tree node that is guaranteed to include them in its scene tree. However, the scene tree approach in this thesis attempts to discard connectivity information as much as possible to reduce storage and processing overheads.

A compromise can be defined such that only a smaller subset of the scene tree's nodes are used as a locale. These chosen nodes have a reference to another auxiliary scene node higher in the scene tree that contains the required subset of nodes for the chosen node. If no other nodes are required, the auxiliary node is the locale node itself. When searching up or down the scene tree, only designated locale nodes are considered.

A function $L_2 = Locale(S, V, L_1)$ is defined that takes the scene S, viewpoint V and current locale L_1 to return the next locale L_2. A second function $F = LocaleEval(S, V, N)$ is also defined to evaluate the suitability of a given locale scene node N to be the locale for V. This function evaluates to a fitness value $-\infty < F < \infty$. A result $F = 0$ indicates perfect fitness as of N as a locale. A result $F > 0$ indicates suitability, but with decreased fitness, proportional to F. A result $F < 0$ indicates that N is not suitable and the unfitness increases proportionally to $-F$.

In the case $F < 0$, a more macroscopic locale is required, or the locale is simply moving to another region of the scene. In this case, the $Locale()$ function must search up the tree for a more suitable locale to introduce a super set of the scene in which to search.

In the case $F \geq 0$, N is a suitable locale. However, there may be a more fit locale in the child scene tree of N that is a subset of N. Therefore, a depth first evaluation of N's child tree is also required, until all paths in the child tree return $F < 0$. If any suitable locales with $F > 0$ are found in the child tree, the option closest to $F = 0$ is accepted as the best.

The incremental search for a locale is therefore defined by having to move up the scene tree if required, from L_1 until a suitable locale L_2 is identified, but then carrying out a

downward traversal from this suitable locale to see if L_2 can be replaced by a better locale in the child tree of L_2.

The pseudo C code in Figure 30 shows this initial upward traversal to find a legal locale in *Locale*(), followed by a downward optimization traversal in *Optimize*(). The search starts with the current scene state S, viewpoint V and locale scene node L being passed to *Locale*(). This function sees if L is still suitable. If it is, then an attempt to optimize L to a better locale in its scene tree will be made immediately. If L is no longer a suitable locale, the algorithm traverses up the scene tree until the first suitable locale is located, which is optimized itself once found. If the scene tree root is reached, it is used regardless of its fitness as a locale, because it is the only scene data available.

The function *Optimize*() searches the given node's scene tree for a better locale with a better fitness value. This search is carried out, recording the best node found, until all paths deeper in the search have unsuitable locales. This search makes the assumption that once an unsuitable locale is found in the scene tree, a suitable locale in its child tree does not exist and the search branch can terminate. For example, a locale may be unsuitable due to position or size and locales in the child tree will only be contained in the unsuitable locale.

The lower bound on this incremental system is $\Omega(1)$, where no change in locale is required. The upper bound remains $O(j)$ for a scene tree with j locale nodes, but is likely to have an average complexity close to constant at times of high coherence.

More complex locale management systems could be formed that use a selection algorithm that results in multiple locales, for example, that represent a series of connected rooms off a central room containing the viewpoint. In this case, a parent locale can be identified that contains all of the required locales, or a temporary scene tree of the locales may be constructed and later discarded. Such algorithms are considered out of scope in this thesis.

FIGURE 30. Simple locale manager pseudo C++ code

```
SceneNode Locale(SceneNode S, Viewpoint V, SceneNode L)
{
        SceneNode       N = L;
        Fitness         F;
                                                // While current locale is illegal and not the scene tree root
        while((F = LocaleEval(S, V, N)) < 0 && IsSceneTreeRoot(N) == false)
        {
                N = GetParentLocale(N);         // Traverse up scene tree to N's parent locale
        }
                                                // Search for better option in N's child tree
        Optimize(S, V, N, N, F);                // with current best node N and best fitness N's

        return N;                               // Return best locale found
}

void Optimize(SceneNode S, Viewpoint V, SceneNode N, SceneNode &NBest, Fitness &FBest)
{
        if(IsLeaf(N))                           // If N is a leaf node, then terminate
                return;
                                                // Get N's child locales
        SceneNode       NA = GetChildLocaleA(N);
        SceneNode       NB = GetChildLocaleB(N);
                                                // Evaluate child node fitness
        Fitness         FA = LocaleEval(S, V, NA);
        Fitness         FB = LocaleEval(S, V, NB);

        if(FA >= 0)                             // If Child node A has legal fitness
        {
                if(FA < FBest)                  // If Child Node A is better than current best
                {
                        NBest = NA;             // Best Node is Child A
                        FBest  = FA;            // Best Fitness is Child A's
                }

                Optimize(S, V, NA, NBest, FBest);       // Search for better option in child A's scene tree
        }

        if(FB >= 0)                             // If Child node B has legal fitness

        {
                if(FB < FBest)                  // If Child node B is better than current best
                {
                        NBest = NB;             // Best Node is Child B
                        FBest = FB;             // Best Fitness is Child B's
                }

                Optimize(S, V, NB, NBest, FBest);       // Search for better option in child B's scene tree
        }
}
```

3.6 Hierarchical View Volume and Back Face Culling

Rendering refinement is carried out from the locale scene node L. Refinement down the scene tree from the locale scene node terminates either when a sufficient level of detail has been achieved, or when nodes are culled. The three main types of culling employed are view volume, back face and occlusion culling. This section examines the first two of these culling methods. Occlusion culling is examined in Section 3.7. Hierarchical culling potentially discards whole sub-trees from the locale L's scene tree. See Section 5.6 for further implementation details.

3.6.1 View Volume Culling

View volume culling makes a major contribution towards reducing the locale L of the scene to be rendered, to a smaller, more manageable subset V, based on the viewpoint's position and direction of gaze (see Section 2.5). All scene detail outside the view volume is discarded as early as possible. Standard rendering systems carry out view volume tests on every primitive entering the pipeline. A hierarchy like the scene tree allows earlier culling at lower levels of detail.

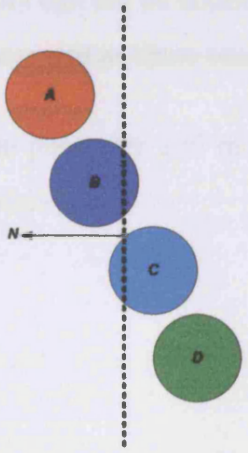
Figure 31 shows four scene nodes with children, each with a different spatial state between a view volume plane and the scene node's spherical scene node. A scene node found to be wholly outside the view volume, such as A can be culled and not refined any more during the rendering of the image. Conversely, image nodes found to be wholly inside the view volume, such as D, need not have any nodes in their child scene tree tested against the view volume, because their parent was found to be wholly inside. Nodes that are found to intersect one or more of the view volume boundaries, such as B and C may still contribute to the image, but their two child nodes must also undergo testing against the view volume.

To generalize, for nodes intersecting or inside the view volume, a state is inherited by the two child nodes that indicates which view volume planes the parent is intersecting. The child nodes only need be tested against those planes. If the parent does not intersect any planes, no tests are carried out in the child nodes. If a child node is not culled, it too passes on its state to its child nodes.

No clipping is required during refinement, because the scene nodes are not necessarily due for rasterization. Image nodes that are rasterized rely on the standard rendering pipeline's culling operations to function on the primitives representing a splat.

The number of view volume tests required for k leaf nodes in L's scene tree has a worst case $O(k)$, where every scene node rendered in the image intersects the view volume. This case is very unlikely to arise in practice and a much better average complexity should be expected. The view volume typically addresses only a small solid angle of the unit sphere and all detail behind the viewpoint should be quickly discarded. See Section 5.6 for implementation details.

FIGURE 31. Scene node view volume plane states



3.6.2 Back Face Culling

Back face culling detects when polyhedral boundary representation surfaces are facing away from the viewpoint, such that they represent the back surface of an object in the second, fourth or greater even depth layer and culls them (see Section 2.6). The scene tree's multi-resolution representation is hierarchical and therefore, a simple normal vector face test used with polygons will not be sufficient.

A back face culling technique is therefore required that caters for the hierarchical, multi-resolution nature of the scene tree, such that a particular scene node can be tested in a conservatively representative way, for its child nodes. For this purpose, we use an adapted normal cone technique similar to the *spatialized normal cone* of Johnson and Cohen [111].

To enable a scene node A to conservatively represent all surface normal vectors of their child scene tree leaf nodes, we store a normal cone C in A that conservatively bounds the normals of the leaf scene nodes in A's child scene tree, with a centroid normal approximation along the cone axis. However, a scene node B in A's child tree with surface normal contained within C can be spatially positioned anywhere within A's spherical volume, with a different view vector from B to the viewpoint.

We can test face visibility for normal cone C positioned anywhere within A's scene node spherical volume (see Section 5.6.2). The hierarchical test distinguishes whether A is entirely *front*, *back* or *mixed* facing. Those front facing are not culled and do not require scene nodes in their child tree to be tested. Those back facing are culled. Those with mixed front and back visibility can not be culled and scene nodes in their child tree must continue to be tested on refinement as these cases lie on silhouette regions.

See Section 5.6 for details on the geometry and maths used in the spatialized normal cone system used in the Canopy algorithm.

3.7 Occlusion Culling

This section gives an overview of the image graph based occlusion culling technique. Occlusion culling detects objects or primitives hidden behind others nearer to the viewpoint such that they do not need to be processed or sent through the rendering pipeline (see Section 2.7). Occlusion culling functions are discussed more detail in Section 5.7.

Hierarchical occlusion culling is employed for two reasons. The first is the conventional task of removing surfaces that are not visible in depth layers after the first visible layer. The second is to define a finite set of visible image nodes that can be used in caching schemes. The occlusion culling method is classed as a *from-point* algorithm [40] which best suits the viewpoint dependent selective refinement nature of the rendering algorithm. The from-point approach is also better suited to dynamic scenes than *from-region* techniques, due the greater emphasis on run-time processing over pre-processing.

Refinement of the image graph and scene tree occurs in a front to back order, with depth limited by level of detail control. To establish if an image node is occluded, there must be information available as to which other image nodes occlude it. The front to back ordering provides this information in the order in which it is required, to prevent the refinement of image nodes that may later be deemed occluded.

The refinement terminates at a minimum level of detail image space threshold t_2, in rendering stage two. Image nodes that achieve this level of detail are added to the *occlusion mask* set M, against which all nodes further away are tested for occlusion. On insertion, a node's occludees are immediately tested. The image graph is used to enforce the front to back ordering and to query which image nodes in the occlusion mask are occluding a particular image node, indicated by the node's image relations. When a child image node is introduced during refinement, if it has at least one occluding image node in the occlusion mask M, it is evaluated for occlusion and is then potentially culled.

Establishing whether a given image node A is occluded or not, is not a straight forward problem. The difficulty lies in measuring how much of it is covered by a set of occluding image nodes. Explicit methods such as rasterization could be used to evaluate this, such as in the Hierarchical Z-Buffer [78] or the Hierarchical Occlusion Map [228]. These

methods are best supported using rendering hardware. The approach described in this section takes a more novel approach based on analysing geometric relationships between the occludee and occluders in image space.

The set of occluders $O \subseteq M$ occluding image node A are at the highest level of detail that will be used for occlusion evaluation. The occludees however, are hierarchical and image node A can be at any resolution between the thresholds for stages one and two, t_1 and t_2 respectively. A single level of detail occlusion mask is used to aid the enforcement of a front to back ordering and to make the system's occlusion evaluation functions more predictable for calibration. Hierarchical occlusion mask adaptations are left for future work.

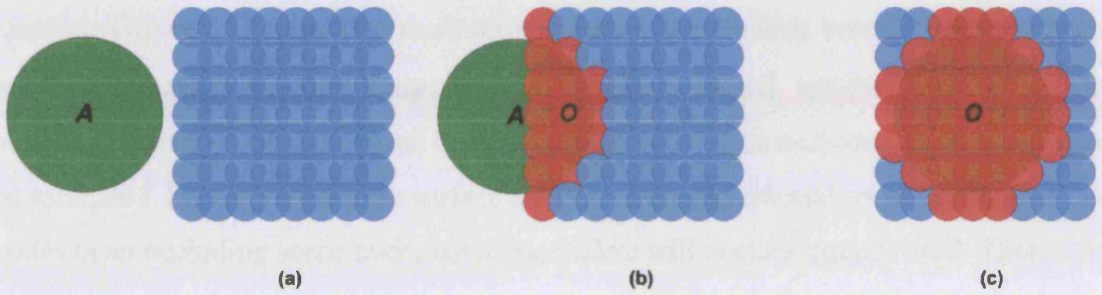
The occlusion evaluation function for an image node uses an approximate classification and not an accurate measurement and therefore is an estimation only. Undertaking the occlusion culling task using specifiable levels of detail for the occlusion mask and variable levels of detail for the occludees, inherently implies approximation and thus potential error. As such, the algorithm may not yet be suited to accuracy critical applications, such as movie effects or medical rendering.

The algorithm must evaluate the aggregate occlusion of image node $A \notin M$, by a set of occluding image nodes $O_A \subseteq M$. The occlusion evaluation classifies A as having one of the visibility states $\{\text{Visible}|\text{PartiallyVisible}|\text{Occluded}\}$. Image nodes that are visible or partially visible must still remain in the image graph. Only those classified as occluded can be culled. The task is therefore to find a way of reliably distinguishing between the occluded state and the partial and full visibility states.

A solution for approximate but acceptable occlusion culling results may not require an exhaustive test of visibility to distinguish the occlusion state and this is one hypothesis that this algorithm investigates. Ideally, if image node A is fully visible in the first visible depth layer, it would have no occluders, shown in Figure 32(a). If A is partially occluded, shown in (b), it must be more distant than its occluders, which lie on the silhouette formed by one or more surfaces in one or more depth layers. If A is occluded, shown in (c), it is must be more distant than its occluders and should receive substantial

coverage, the extent to which is a function of the distribution of the occluding image nodes in image space and how well they occlude based on the distribution of the surfaces that they approximate. Classifying A as visible or occluded is likely to be simpler than the in between case of partial visibility. As such, the most likely errors are predicted to be blank halos around silhouettes of nearer surfaces, due to occluded image nodes further away that have been incorrectly classified as occluded, rather than partially visible. Any degree of conservative classification as partially visible rather than occluded, may be preferable, subject to the node's presence not leading to disruptive culling behaviour as the refinement progresses through the scene.

FIGURE 32. **Visibility states (a) *Visible*, (b) *PartiallyVisible*, (c) *Occluded***



To evaluate A's degree of occlusion, a summation and thresholding scheme is used, where each of A's occluders in O make an *occlusion contribution* to A individually. This contribution takes no consideration of multiple contributions of overlapping occluder regions, because the calculation of the intersection of the occluder image nodes O with each other and A is computationally expensive. The total occlusion value T is then compared with a threshold $t_{OccCull}$ and culled if $T > t_{OccCull}$. It is assumed that an average amount of overlap will exist and be absorbed by an increased $t_{OccCull}$ threshold, which is empirically calibrated. The occlusion values of states (b) and (c) in Figure 32 are typically well distinguished for two reasons. Firstly, in (b), node A receives fewer occlusion contributions due to fewer occluders than case (c). Moreover, a higher proportion of contributions in (b) are from occluding nodes near the silhouette and will contribute lower values due to lower occlusion strength. Due to the intersecting nature of scene nodes to form contiguous surfaces, image nodes that should be visible, may be occluded by neighbouring image nodes, causing self occlusion in surfaces, particularly when viewed at grazing angles. The algorithm currently avoids methods such as discarding

contributions based on surface IDs to solve this problem, as this would introduce additional complexity and is not straight forward in the case of the hierarchical scene tree representation. To reduce this problem, image nodes that have scene nodes that are intersecting in object space, do not make occlusion contributions to each other. Overlap areas of nodes in the same surface at further distances than a single node should be smaller and less affected. Future work may focus on additional techniques such as ramping up contributions with increased object space distance, to relieve this problem.

Several functions are defined. The primary function $OccEstimation(A)$ evaluates the aggregate occlusion of image node A by all n image nodes in A's occluding set O. This function adds occlusion contributions made by each occluder $B \in O$ using the function $OccContrib(A, B)$. The function $OccCull(A, t)$ uses a thresholding comparison to return a boolean that states whether image node A should be culled, against threshold t. The occlusion contribution made by an occluder image node, to an occludee image node must be estimated. Due to variation in surface area, position and orientation of leaf level scene nodes in an occluding scene node, not all occluders will occlude equally well. Therefore, a view dependent function $OccStrength(B)$ is defined to evaluate how well the image node's scene node can occlude. The level to which the occluder covers the occludee must also be known and is given by $OccRatio(A, B)$ that calculates the ratio of occlusion, as the area of intersection of A and B's viewing solid angles, normalized and clipped to $[0...1]$.

The function $OccStrength(B)$ is view dependent. This is implemented using a method we've termed the *parallel visible area* function $PVA(B)$. This function uniformly distributes the scene node's total surface area over the solid angle of its normal cone. It then calculates how much of this occluding surface area is component to the viewpoint, based on the direction of view and divides this visible area over the cross sectional disc area of the occluder. Occludee image nodes behind silhouettes will have lower total occlusion values because they have fewer occluders. Moreover, each of these occluders lie on the silhouette and will make lower occlusion contributions because less of their surface area is component to the viewpoint, making their occlusion strengths weaker. Occludees within the interior of an occluding surface will receive a greater number of occlusion contributions, with higher strength values. To increase accuracy at silhouettes for vistas, occluders can also be clipped at low strengths such that they make no contributions.

3.8 Rasterization

This section discusses the rasterization in stage four that finally draws the set of image nodes resulting from stage three to the frame buffer.

The image graph resulting from stage two represents a set M of image nodes that are considered visible or partially visible from the viewpoint. However, their refinement is limited to level of detail t_2, specified as an image space minimum size threshold. This level of detail should be sufficient for occlusion mask functionality, but may not be suitable for rasterization. Stage three refines each node $m \in M$ to a set $R_m \in R$ of image nodes at a higher level of detail threshold t_3 that is suitable for rasterization by stage four. The refinement in stage three does not update the image graph, for increased speed.

Each image node is rasterized using splatting, a technique introduced by Westover [211] that has been used widely in the volume rendering field and in the QSplat [171] and Surfel [158] point based rendering systems (see Section 2.9.5). Splatting rasterizes an image space primitive that represents the required footprint.

In the case of an image node's spherical scene node, an open viewing cone is constructed with its apex at the camera's center of projection (see Section 1.2) whose sides are tangental to the sphere. The viewing cone intersects the camera's view plane, forming an elliptical conic section footprint. A common misconception of the elliptical conic section is that the intersection boundary is egg shaped, but it is in fact a perfect ellipse.

The elliptical conic section would be an acceptable splat shape, but is slow to calculate in software, although pre-rendered elliptical bitmaps could be used. Hardware graphics pipelines generally do not support the rendering of 2D ellipses. An alternative scheme would be to use a circular textured support polygon with an alpha map for transparency that will give the required profile under a perspective projection, but the supporting polygon is slow to construct.

Considering that scene complexity is increasingly high and that current and future hardware systems may be capable of refining to near pixel, pixel or sub-pixel levels of detail in all frames, splatting may be of decreasing importance and pixel level operations may

be sufficient, without the standard 3D rendering pipeline. Where detail is not available, procedural generation may be able to enhance geometrically.

Therefore, we do not place a strong emphasis on the correctness of the splat and use an approximation. The system first calculates the conic section as an ellipse with semi-major and semi-minor axes and approximates it using a graphics hardware point primitive with radius equal to the semi-major axis to conservatively bound the ellipse. Details on the conic section and point size calculation are given in Section 5.8.1. The end result is dependent on the pipeline used, which can range from a square or rounded square to a disc.

Modern graphics processing units (GPUs) have vertex processors that can execute vertex programs. Such a program can be easily written in Cg/HLSL [60] that calculates the conic section in hardware to exploit the speed and parallelism offered.

Depth ordering is required to resolve occlusion between overlapping splats that are rasterized. Depth ordering for rasterization can be enforced at pixel resolution, using the standard z-buffer technique [63], or without a z-buffer at level of detail t_2 using the reverse of the front to back ordering implied by the totally ordered set M, resulting from stage two. Each node in $m \in M$ is refined in front to back order in stage three, to give a set of sets where $R_m \in R$. The ordering between sets in R reflects that of the ordering of each parent m in M. The ordering within each set R_m is random. Therefore, the sets $R_m \in R$ are rasterized in reverse order in R to achieve a back to front compositing order.

This approximate depth ordering technique will improve as the occlusion mask level of detail threshold t_2 approaches pixel sizes. It is also better suited to scenes where each set R_m has only one front facing depth layer, as is most likely in small surface regions of models in a scene. The approximation is not suitable for accuracy critical applications such as movie effects and medical systems, but may be suitable for games or other visualizations where some degree of error is tolerable.

It is important that the arbitrary rasterization ordering within each R_m is maintained between frames to prevent changes in depth ordering that would give rise to noticeable artifacts due to changing occlusion between rasterized splats.

The system currently does not include explicit pre-filtering for anti-aliasing. To help reduce aliasing, we simply do not let splat sizes become substantially sub-pixel, so as to band limit the surface signal and encourage raster sampling above the Nyquist frequency. Hierarchical surface attribute approximation, weighted according to surface area, also has a mip-map like effect to low pass filter surface frequencies for rasterization.

Local illumination shading is executed for each image node in R. While nodes in R are only a potentially visible set, the large majority of image nodes should be visible with minimal over-draw. Each splat receives a single result based on a set of point light sources defined in the scene. Alternatively, shading can be evaluated for image nodes at lower levels of detail and inherited by their child tree of image nodes, but modern graphics hardware is generally capable of achieving full shading for each splat, particularly because the amount of over-draw in the system is limited by the occlusion culling system. Per fragment shading can also be achieved using a GPU splatting fragment program [60] (see Section 2.9.5.7) but an emphasis is placed here on achieving near pixel or pixel level primitives for rasterization.

Topics in Section 5.8 cover specifics of the conic section based splatting system used in the Canopy algorithm.

3.9 Image Tree Caching

An optional caching scheme is used that stores the image nodes resulting from the refinement of multiple frames over a period of time. The cache is used to store viewpoint independent or dependent information that may be reusable in subsequent refinements. It is also used as a basis for hierarchical shadow mapping using an image tree tracing technique described in Section 4.1. Each image node is the image space analogy of its scene node in the scene tree. As an image node is refined, two new image nodes are introduced that represent the child scene nodes. The image tree simply stores image nodes for reuse, in a tree that reflects the structure of the traversed regions of the scene tree. The rendering algorithm *traces* the existing image tree as the refinement progresses, to locate reusable image nodes. As an image node is refined by the rendering algorithm, the image tree is first checked to see if the child image nodes are present. If child image nodes are in the image tree, the image node data structures are reused, with data contained within them selectively reused. If the image nodes are not present in the image tree, new image nodes are created and added to the tree. Image nodes that are culled are tagged and kept in the image tree for algorithms that require this information such as shadow mapping or other potential data reuse schemes. Over the course of a sequence of frames, the viewpoint may move and introduce new branches into the scene tree that were previously not present.

Over time, the image tree will grow to represent surfaces in and out of view. To keep the image tree at a maintainable size, a pruning algorithm is used based on a least recently used image node policy.

The algorithm caches and reuses decompressed viewpoint independent scene node geometry stored in each image node. Future work may research the caching of viewpoint dependent image node information to exploit inter-frame coherence when changes are detected that are small enough to allow reuse without incurring substantial error. Such candidates could be the viewing cone, normal calculations or culling states. Future caching variations may be investigated, particularly to reduce bandwidth between the CPU and GPU in each frame, e.g. based on caching rendering refinement stage three points in a way similar to Sequential Point Trees [48] (see Section 2.9.4.3).

Occlusion culling is an important factor for the image tree system, because it defines a smaller set of visible image nodes for caching. Without it, the image tree would be far larger and may not be a suitable technique. If the locale node changes between subsequent frames, currently the image tree is deleted and reconstructed for the new locale. A more efficient system is to use the subset of the existing image tree defined by the new locale if it is within the existing image tree, or include the existing image tree in a new image tree if the new locale is higher in the scene tree than the existing locale.

Hierarchical shadow mapping from point spot light sources can be achieved by exploiting similarity between occlusion culling and calculating shadows by rendering the locale from each light's position (see Chapter 4). The image tree for each light source represents a multi-resolution shadow map. Light source image trees are traced simultaneously when rendering the camera's view, to determine which light sources are illuminating the scene nodes of image nodes in the camera's image tree.

3.10 Parallelization and Pipelining

Parralelization and pipelining issues must be addressed at both the algorithm design and implementation stages. The algorithm lends itself to parallel computation and a pipelined architecture for efficient implementation, to distribute across multiple processor elements (PEs) or custom modules in hardware. Parallelization allows the same or different tasks to be carried out simultaneously. Pipelining allows different tasks to be carried out simultaneously, each taking input from a pipeline module and passing the output to another pipeline module.

To see how we can separate the algorithm into appropriate modules, we must identify similar tasks that are performed iteratively and differing tasks that could be performed simultaneously. Here, we will only look at the algorithm parts that can be separated, without giving concern to any specific types of hardware platform, such as multi central processing unit (CPU) systems or mappings onto graphics processing units (GPUs).

In stages one to three, image nodes are refined. In stage one, the order can be depth or breadth first, until level of detail t_1 is reached. The image graph must be updated during this refinement and passed on to stage two. Stage two then refines using the front to back ordering, with view volume, back face and occlusion culling. The resulting refined image graph is passed on to stage three, where its image nodes are refined quickly, to higher resolutions for rasterization.

Image node creation and refinement lends itself reasonably to parallelization. In stages one and three, there are large sets of image nodes to be refined. In stage two, there is a set of order invariant nodes, that could all be refined at once, although image graph relations affect two nodes, so mutual exclusion on image nodes may be required to prevent two image nodes undergoing operations simultaneously. Stage three's refinement is fast and independent of the image graph.

Tasks that may be parallelized include the calculation of view dependent data in image nodes, the evaluation of image graph relations, view volume culling, back face culling and occlusion culling. Rasterization can also be carried out in parallel and indeed is, when a modern hardware rendering system is used.

Stage one can be refined in parallel, but due to the ordering required in stage two's image graph refinement, it is likely that tasks in stage two could not form a later module in a pipeline and therefore, stage two tasks may need to wait until stage one's refinement is complete.

Stages two to four lend themselves to pipelining. In stage two, if an image node has level of detail t_2, it can be passed on to stage three. Stage three refines the node independently to a larger set of image nodes at level of detail t_3 in depth first order. If an image node has level of detail t_3, it can be passed on to stage four for rasterization.

This pipeline may not have uniform timings, but will allow separate modules to perform specific tasks simultaneously. Each of these modules could exploit parallel execution internally.

3.11 Algorithm Complexity

The simple, incremental locale management algorithm's lower bound is $\Omega(1)$, where no change in locale is required. The upper bound remains $O(j)$ for a scene tree with j leaf nodes, when every node in the tree is a potential locale and is evaluated, but is likely to have an average complexity close to constant due to temporal coherence resulting from conventional navigation styles.

Refinement from a locale scene node to a finite level of detail in image space has lower bound $\Omega(1)$ in the case where the locale has sufficient LOD and refines a maximum of l nodes at the suitable resolutions chosen by the level of detail control system. If $l < k$, where k is the total number of leaf nodes in the locale, the complexity is dependent on l and independent of k and is thus output sensitive. Aggregating LOD may also combine and limit the number of depth layers in k used, such that the algorithm may be independent of the depth complexity.

When view volume, back face and occlusion culling are introduced, the number of terminal nodes addressed i has a lower bound $\Omega(1)$ when the locale node is culled and addresses a maximum of l nodes when $i = l$, such that all nodes of suitable LOD are tested and not culled. Otherwise, the number of terminal nodes addressed is $i = r + c$ where r is the number of un-culled nodes c the number of culled nodes. The number of branching nodes addressed is $i - 1$, totalling $2i - 1$ addressed nodes. Analysis of an average running time due to culling may be possible using probabilistic modelling but is very difficult due to the varying types of scenes that can occur and will not be given here.

However, introducing image graph refinement to the discussion, shows a non-linear worst case. Refining the occlusion mask to i nodes has a best case when no image nodes overlap in image space. In this case, only one relation test is carried out per branching node, to establish any relation between child nodes. This case therefore tests a total of $i - 1$ relations during refinement, forming a linear lower bound $\Omega(i)$ for image graph refinement. A worst case number of relation tests is undertaken when all image nodes overlap in image space throughout the hierarchy, for example, when looking down a row of leaf scene nodes. Each node refined introduces exactly one new active node in total,

with the previously active refined node becoming inactive. Given an occlusion mask with n active nodes, when refining one single node, the number of relations tested and formed due to this action is $1+2(n-1) = 2n-1$, that accounts for the single relation between the new child nodes and $(n-1)$ relations formed between each child and the remaining active nodes, therefore $2(n-1)$. Given a worst case refinement of the image graph from a single root node to i nodes, a total of $i-1$ refinements are required. The total number of relations formed during the history of this refinement is:

$$\sum_{n=1}^{i-1}(2n-1)$$

$$= 2\left(\sum_{n=1}^{i-1} n\right)-(i-1)$$

$$= 2\left[\frac{(i-1)^2+(i-1)}{2}\right]-(i-1)$$

$$= (i-1)^2 \qquad \text{(EQ 8)}$$

Thus, image graph refinement has a worst case complexity $O(i^2)$. With respect to the scene tree, if $i = l = k = j$, where the locale is the root node of the scene tree, all leaf nodes have suitable LOD, no culling is effective and a highest LOD is used for image graph refinement (thus making stage three refinement redundant), a total of j nodes will result. The algorithm's worst case complexity is then $O(j^2)$. Any further depth first refinement in stage three would add a typically linear overhead $O(i)$.

In practise however, the algorithm is likely to exhibit far better scalability because typical situations are far from the worst case. The image graph of most scenes will have a more even distribution over the image, reducing the number of relations processed per image node.

When rendering a very large, highly detailed scene with well distributed features, it is likely that $i < l < k < j$. It is likely that the locale management will incrementally identify a new locale in near constant time, and $k < j$ by substantial magnitudes. The levels of detail

required will be such that $l < k$ significantly. With back face culling, about half of the leaves in l should be discarded. View volume culling will discard a large proportion of l. In highly occluded environments, depth layers behind the first visible layer are culled earlier. In general, it can be assumed that $i < l$ by a significant magnitude. Rendering termination to terminate refinement when the image is considered complete is left as future work, but would also serve to reduce the number of occlusion culled nodes significantly in scenes with high occlusion.

Summarising complexity dependence in practical use, locale management to identify k scene nodes in the locale can be independent of the number of scene nodes in the whole scene j. On refinement, with level of detail control, the number of image nodes in the occlusion mask i and subsequently refined in stage three is independent of the number of nodes in the locale k. Aggregating LOD may also combine and limit multiple depth layers in the view volume such that i is independent of the depth complexity d.

3.12 Summary

This chapter began by looking at a number of key tasks in a framework to achieve sub-linear scalability and break complexity dependencies on data size. Specifically, these tasks are *locale management*, *view volume culling*, *depth ordering*, *back face culling*, *occlusion culling*, *level of detail control* and *rendering termination*.

A scene representation was then identified to suit these tasks, based on a uniform, scale independent form that unites the concepts of object and primitive, such that algorithms working on the data need draw no strong distinction. The rendering algorithm was also designed, in co-operation with the form of scene representation.

A high level view of the algorithm was given, with examples of how an image graph can be used to determine front to back traversal order and support occlusion culling. The four main rendering stages were explained. These rendering functions are detailed more thoroughly in Section 5.4. Refinement termination was discussed, required to reduce dependency on the depth of scene data in the view volume after the image is complete.

The scene tree representation was then examined, with a description of a hierarchical spatial partitioning system based on BSP / K-D trees. Scene tree construction will be looked at again in Section 5.3.

Locale management was discussed as a means for rapid culling of scene data, to define a smaller *Locale* to be solely used for rendering. A basic algorithm is given for incremental distance based locale identification to act as a root scene tree node for rendering.

Hierarchical view volume and back face culling were examined to reduce the locale to a subset contained in the view volume and to remove the rear sides of visible polyhedra. Further geometry and maths are covered in Section 5.6.

Occlusion culling using the image graph was examined, which is further detailed later in Section 5.7.

Rasterization based on splatting was examined with approximate depth ordering that will be further detailed in Section 5.8.

The rendering algorithm shares sub-tasks and data. For example, the image node view cone is used for hierarchical back face culling, occlusion culling, level of detail control and splatting. The scene node normal cone is used in hierarchical back face culling, occlusion culling and shading. Scene node surface area is used for scene tree weighted colour filtering and occlusion culling. The image graph data structure provides front to back ordering and geometric coverage information for occlusion culling and a back to front ordering for rasterization.

Image trees were discussed in the context of caching decompressed geometry for reuse in subsequent frames and serving as a basis for additional caching schemes. The next chapter will discuss how the image tree can also be used for a form of hierarchical shadow mapping.

Restructuring of the algorithm was discussed to achieve pipelined and parallelized architectures.

Finally, the algorithm's complexity bounds were overviewed. Whilst the algorithm has a worst case upper bound of $O(j^2)$ where j is the number of leaf nodes in the entire scene, the practical complexity is likely to be far better when viewing a typical scene, because image nodes are more evenly distributed over an image, with a mean number of relations per node and will later be demonstrated to be sub-linear. Locale management may be independent of the number of scene nodes in the whole scene j and refinement may be independent of the number of scene nodes in the locale k. Due to surface aggregating LOD, rendering may be independent of the number of depth layers d in the view volume.

Chapter 4

Shadows and Collision Detection

This chapter introduces additional features that exploit and demonstrate the versatility of the scene representation and rendering systems introduced in Chapter 3.

Hierarchical shadow mapping is overviewed in Section 4.1, using a technique we term *image tree shadow tracing*.

Hierarchical collision detection techniques are then given in Section 4.2 for point-object collisions and object-object collisions.

Lastly, a ***summary*** can be found in Section 4.3.

4.1 Hierarchical Shadow Mapping

Shadows are cast on a surface in a scene due to a lack of line of sight visibility with an emitting source. For point spot light emitters, the process of calculating which surfaces are in shadow and which are not is very similar to the occlusion culling task for rendering a scene from a viewpoint. Slater, Usoh and Chrysanthou indicate that shadows offer an increased sense of presence in visually dominant subjects [192]. This section will examine how the algorithm can be used to calculate shadows from a point spot light source. Previous work on shadows is extensive and the reader is referred to surveys by Woo and colleagues [217], Haines and Moller [90] and Hasenfratz and colleagues [93].

Point source emitters do not occur naturally, but are widely adopted as an acceptable approximation in graphics, particularly for realtime rendering. They are not capable of producing the effects of area light sources found in real scenes, as calculated in other methods such as radiosity or ray tracing with area light source sampling. As such, no penumbra will be visible within cast shadows, which will be quite sharp and uniform in shadowed areas. Multiple renders can be used to sample an area light source, but this will not be considered as it is an extremely expensive option computationally for realtime systems.

The next sections will discuss how basic hard shadows can be calculated and rendered using the existing image graph system with image trees.

4.1.1 Shadow Calculation Using Occlusion Culling

The standard camera model can be used to describe the basic properties of a point spot light. The position and view plane normal (VPN) vector are primary controls for positioning and orienting a light source (see Section 1.2). The view up vector to set the camera's *roll* is less important, particularly if the light's intensity has a radial fall-off function from the view plane normal, such that the frustum may not define the cast shape.

For each light source in the scene, a separate rendering process is carried out with the camera configured by the light source's parameters, before the viewpoint image is rendered. The result of each light source's rendering is a hierarchical map of which surface

regions are visible to the light source, along with an approximate measure of the degree of occlusion. These results are then correlated with the surfaces visible in the standard viewpoint rendering. This process is similar to shadow mapping, initially developed by Williams [214] based on projective texture mapping described by Segal and colleagues [183], now commonly implemented using GPUs [58]. Shadow mapping in a hybrid point based system is described by Guthe et al. [88]. Although the method presented here is not a buffer based method, it is referred to as a shadow map due to the similarity of the information it embodies and of the general process. The method is subject to the same sampling issues that cause reduced resolution of the shadow boundaries over distance from occluding surface and the occludee due to the discrete sampling.

Only a partial render is required, using stages one and two. The extra refinement and rasterization in stages three and four are not relevant because the shadow calculation is only concerned with determining visibility at some level of detail (see Section 3.3.7 for details on each rendering stage).

If shadow calculation is to be correct, the occlusion culling solution must be accurate enough to detect and cull surfaces occluded by a single occluding surface. In the case of the image graph system, it can not be conservative in its culling thresholds and must be responsive to just a single depth layer to detect shadow volume containment. As will be seen in Chapter 6, it has been found that the algorithm can detect occluded regions with just a single depth layer, with what is likely to be an acceptably low level of error.

A primary difference between occlusion culling to render an image and occlusion culling for shadows is that in image rendering, the occlusion culling process in stage two can terminate at lower levels of detail. In the case of refinement in stage two for shadows, the process must continue to higher levels of detail to result in sharp shadow boundaries. The occlusion culling stage will therefore take longer when calculating shadows than viewpoint images.

Shadows need to be updated if either the light source or objects in their view volume move. If both are static between frames, the previous results can be used to render a viewpoint image. If the view volumes of the light source and the viewpoint do not intersect, the light's shadows do not need rendering when using a local illumination model.

4.1.2 Light Source Image Tree Shadow Tracing

Images are rendered from each light source using rendering stages one and two using the same locale scene node. This results in an image tree and image graph for each light source. A light source's image tree is a hierarchical record of the refinement during the occlusion culling process. The leaf image nodes of the image tree represent the set of nodes at the highest refined level of detail that are visible (illuminated), or those that are culled at some level of detail (shadowed), which are flagged as culled.

The image nodes from each light source's rendering must be correlated with the image nodes that are visible in the viewpoint rendering, to establish which light sources are illuminating the image node's scene node. This is carried out using a method that will be termed *image tree shadow tracing.*

Consider the image tree of the viewpoint undergoing refinement and that of the a light-source that has already been generated. Because they originate from the same scene node, their branching structure also describes the same scene nodes with additional view dependent information present in the image nodes.

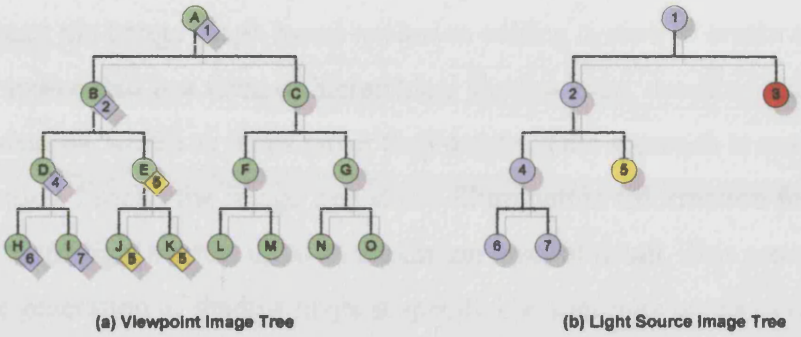
Each image node in the viewpoint's image tree stores a set of image node references, one for each corresponding non shadowed image node in each light source's image tree. The root image node of the viewpoint's tree is initialized before rendering, with an entry for each light source to be considered. When a viewpoint image node is refined, each reference is checked to see what will be inherited by the two child image nodes. Three possible cases exist that affect this inheritance:

1. Child light source image node is not culled
2. Child light source image node is culled
3. Child light source image nodes do not exist

Each case is present in the simple single light source trace example shown in Figure 33. The viewpoint image tree (a) has image nodes A to O with traced node references denoted by references 1 to 7 from the light source tree (b). Light source image node 3 is culled. Image nodes 5, 6 and 7 are simply leaves.

Several cases are shown in this example. During the viewpoint's refinement, image node references have been traced from (A, 1) to (B,2), (D, 4), (E, 5), (H, 6) and (I, 7).

FIGURE 33. Viewpoint image tree with single light source image tree trace



(a) Viewpoint Image Tree (b) Light Source Image Tree

In the first case, image nodes H and I have traced down to light source image nodes 6 and 7. In this case, the light source image tree has sufficient level of detail to match the viewpoint refinement. The light source's image tree may even continue to deeper levels, but will not be used in the viewpoint's rendering due to insufficient supporting geometry.

In the second case, image node 3 in the light source's image tree has been found to be culled, i.e. not visible and therefore in shadow. In this case, no illumination entries are received by viewpoint node C or its child tree.

The third case handles situations where the viewpoint's image tree refinement advances to higher resolutions for some scene regions than that of the light source. In this case, the image nodes in the viewpoint tree must still receive illumination and therefore both inherit the same light source leaf image node. An example is shown in the refinement of E where nodes J and K receive leaf node 5.

Once the viewpoint's image tree has been completed during rendering, each image tree leaf node will contain a set of references to image nodes pertaining to each light source that illuminates it. If no references are present, no light source illuminates the image node and all that will remain to shade it is the simple ambient term common to most local illumination models.

Nodes H, I, J and K are illuminated by the light source, which is used in the local illumination shading model. Image nodes L, M, N and O are in shadow and have no illumination contribution from the light source. If light source image trees have a sufficiently high level of detail, their occlusion values may be used to linearly modulate shading to give a softening effect on shadow boundaries.

4.1.3 Properties of Image Tree Shadow Tracing

The system uses the image graph based occlusion culling system to create an image tree for the light source that is a form of hierarchical shadow map, detailing regions that are in light and shadow, stored as states rather than depths. This approach is multi-resolution in several senses. Firstly, the image tree stores illumination information for each scene node visible to the light source, up to its maximum level of detail. This provides a mechanism for the generation of shadow maps at specifiable minimum levels of detail to trade off between time and quality. The shadow receiving geometry is also multi-resolution and need only make use of as high a level of detail of the shadow map as it requires during viewpoint rendering. If the receiving geometry requires higher levels of shadow map detail than are available, the maximum but lower level of detail shadows are inherited and represented by the higher level of detail receiving geometry.

Sattler and colleagues [176] have shown through experiments with polygon based models, that up to 99% of an shadow casting object's geometry can be decimated before 90% of subjects notice unsatisfactory shadow shapes.

4.2 Hierarchical Collision Detection

The hierarchical nature of the scene tree offers potential for fast, logarithmic time collision detection. Point-object and object-object collision detection are examined here, to further explore the versatility of the same hierarchical data structures used for rendering.

Applications commonly require collision detection between the viewpoint or point of interaction and the scene to prohibit movement through surfaces. Collision detection between objects is also valuable in some applications, for example in 3D modelling user interfaces or physical simulations.

Many collision detection algorithms have been developed for various forms of geometric representation. Surveys of collision detection techniques are given by Lin and Gottschalk [131] and Jimenez and colleagues [110]. Hierarchies are common to collision detection solutions. In particular, hierarchies of bounding spheres have been used by Hubbard [106], [107], Quinlan [162], Bradshaw and O'Sullivan [23] and Mendoza and O'Sullivan [147].

This section will present two very simple methods for point-object and object-object collision detection. They are based on whatever underlying scene tree hierarchies are constructed and are not concerned with attempting to achieve accurate or efficient hierarchies for collision detection use, such as the medial axis method Bradshaw and O'Sullivan [23]. The balancing of these trees will affect the efficiency of the collision detection in practice, but an assumption is made that the scene tree is not too skewed.

4.2.1 Point-Object Collision Detection

Application navigation systems attempt to move a viewpoint from one position to another between frames. Interaction devices with force feedback, such as the Phantom [145] may move interaction points from one location to another. Physical simulations such as cloth also move points located on the surface of the material. If a collision occurs during this motion, it must be detected and the position and surface normal supplied to support a response.

A point's motion can be considered linear if moves are fairly small, without significant error in practice. The task of detecting whether the viewpoint passes through an object

can be both simple and fast. A basic form may be used that simply calculates collision based on the legality of the proposed end point of the motion vector, though this method is only suitable over small distances if it is not to miss collisions along the motion vector. This section will discuss how viewpoint-object collision can be carried out using either end point or motion vector tests.

A candidate set C is created that initially contains the locale scene node. Items in the candidate set are successively removed and tested for proximity with the vertex or motion vector's closest approach to the test node. If they are within specified proximity, a collision is recognized. If they are sufficiently distant, no collision occurs and no further refinement of the test node occurs. Otherwise, refinement of the test node continues with the test node's children added to C.

Two criteria are specified. The first is a surface level of detail threshold T_r. The second is a collision distance threshold T_d from the scene node surface to the test point. Together, these specify a maximal tolerance distance $\varepsilon = 2T_r + T_d$ around the test point. A collision is guaranteed and recognized if $|X-P| + r < \varepsilon$, where X is the test point and P and r are the test scene node's position and radius. Otherwise, if $|X-P|-r < \varepsilon$, a collision may occur but is not guaranteed and the node is refined and its children placed in C, else no collision occurs. If the test node is a leaf and $|X-P|-r < \varepsilon$, a collision is recognized. Refinement may continue, to recognize a single nearest collision, multiple collisions or the process may terminate when the first is found, for speed.

Collision response systems specify a new end point, based on the rejection of the end point proposed by the client algorithm. Currently, a simple rejection is used that reuses the previous frame's position, but the end point could be interpolated to a more accurate collision position.

4.2.2 Object-Object Collision Detection

Collisions between objects can be detected using a method based on pair tests. The method discussed here is similar to that of Quinlan [162] and Mendoza and O'Sullivan [147]. Like the point-object collision parameters, level of detail and distance thresholds T_r and T_d are specified. Multiple collision points can result between objects in each frame.

Starting from the locale scene node, its two child scene nodes are added as a pair to an empty, ordered candidate set Z to set up the algorithm. A pair $z = (A, B)$ where $z \in Z$ is iteratively removed from Z and tested for level of detail (size) and against each other for proximity. If the spheres in pair z are intersecting or their distances measured between their surfaces are less than or equal to threshold T_d and their level of detail is too low resolution, refinement continues. The non leaf node with the largest radius in z is selected for refinement to minimize the total sizes of candidate nodes (e.g. A to child nodes C and D). Two new pairs (C, B) and (D, B) are added to the end of Z containing the unrefined node and one refined child of A each. If a pair's separation is shown to be further than the T_d, the pair is discarded.

Pairs that satisfy both the level of detail and distance thresholds are added to a result set R. The process terminates when $Z = \{\}$. The resulting set R can contain multiple points of collision. Alternatively, the process may terminate when the first collision is found. The node's position and normal cone can be provided for the collision response.

4.3 Summary

In this chapter, we've looked at *image tree shadow tracing,* a simple hierarchical shadow mapping style technique for hard shadows that establishes a multi-resolution shading status from point light sources, represented by an image tree. Rendering stages one and two are run from the point of view of a directed point light source or spot light. The resulting image tree describes image nodes in the locale that are illuminated. Leaf image nodes in the tree that are culled are not illuminated and this state can be inherited by their child trees. Rendering then uses image tree shadow tracing to establish illumination information of visible image nodes from the viewpoint and inherit shaded states.

Point-object collision can be achieved in a very simple hierarchical algorithm, that incorporates both specifiable proximity and level of detail constraints.

Object-object collision detection can also be easily achieved, using a system similar to Quinlan [162] and Mendoza and O'Sullivan [147] that also incorporates specifiable level of detail and proximity constraints. Candidate pairs of scene nodes are successively tested in object space, refining scene nodes whose child trees may contain satisfying collision candidates, rejecting pairs that do not and accepting pairs that meet the criteria as one or more points of collision.

Chapter 5 *Implementation*

This chapter describes the algorithm's implementation in more detail. Additional information about the system's architecture and development issues can be found in the appendix. Firstly, this chapter will look at the ***scene node*** data structure in Section 5.1 and how scene nodes can be ***sampled*** from polygonal scenes in Section 5.2. Details on how the hierarchical ***scene tree*** is constructed are given in Section 5.3.

The main ***rendering stage functions*** are examined, with ***pseudo code*** in Section 5.4.

The ***image graph*** data structure is described in Section 5.5, detailing how it is updated during refinement to provide information for depth ordering and occlusion culling. Hierarchical ***view volume culling*** and ***back face culling*** geometry are examined in Section 5.6, moving on to details of how the image graph is used for ***occlusion culling*** tasks in Section 5.7. ***Rasterization*** based on ***Splatting*** is then covered in Section 5.8.

Additional aspects of ***geometric compression*** and ***scene graph representation*** with compression are given in Section 5.9 and Section 5.10. Finally, the chapter discusses the need for ***hierarchical consistency*** and potential for ***scene dynamics*** in Section 5.11.

A ***summary*** of the chapter is given in Section 5.12.

5.1 Scene Node

This section details the attributes stored by the scene node (see Section 3.4.1). Storage and compression issues are discussed in Section 5.9.1. The attributes detailed are *position, radius, normal cone, surface area* and *diffuse colour*.

5.1.1 Position and Radius

Position and radius are used to conservatively bound the volume of the original surfaces. Leaf scene nodes already have position and radius defined by the sampling process, that should ensure contiguous connectivity of surfaces, without holes. The position and radius of scene node points at internal branching nodes in the scene tree are calculated during the scene tree's construction. If a scene node has position P with radius r, the parent sphere of two child spheres which are separate or intersecting but not containing can be simply calculated as follows:

$$V = P_B - P_A$$

$$l = |V|$$

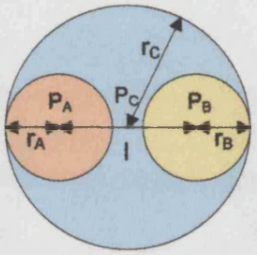
$$F = \frac{V}{l}$$

$$P_C = \frac{P_A + P_B + F(r_B - r_A)}{2} \quad \text{(EQ 9)}$$

$$r_C = \frac{l + r_A + r_B}{2} \quad \text{(EQ 10)}$$

Given two child nodes A and B, the parent C encompasses the span of the two child spheres, shown in Figure 34. The vector V between A and B is calculated, along with its length l which is then used to calculate a normalized axial vector F. This vector is then projected outwards from child positions P_A and P_B with their respective radii along the axis vector. The position of the parent P_C given by (EQ 9) is then the mean of the two end points of the span. The parent radius r_C is half the diameter formed by l, r_A and r_B given in (EQ 10). The parent then has the tightest fit possible for two spheres.

FIGURE 34. Parent sphere C's position & radius defined by child spheres A and B



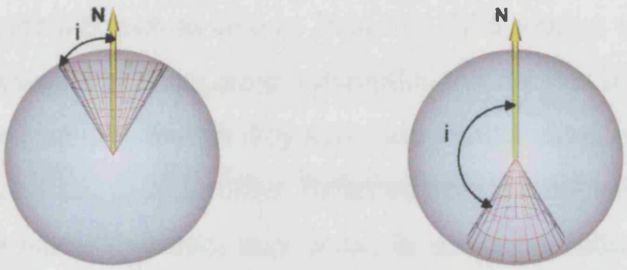
A special case exists where a child A contains B, at which point the parent is equivalent to A. The special case where $P_A = P_B$ and $r_A = r_B$ can be treated similarly where the result is equivalent to A or B. However, when first constructing hierarchies from sampled scenes, contained or equal samples either do not exist due to the design of the process that generates them, or are removed for efficiency and so do not occur.

5.1.2 Normal Cone

A surface normal N is stored for each scene node that approximates those of the higher resolution nodes in its scene tree. A normal cone measured from N where $i = [0...\pi]$ is defined such that its cone contains all normals of all scene nodes in its scene tree, shown in Figure 35. This method has also been elsewhere in the literature, including QSplat [171] [172]. Note that the cone can exist as an acute angle cone shown in (a) or an obtuse inverted cone (b) to represent any extent of normals up to and including the unit sphere of normal vectors. The normal cone is used in both back face culling and occlusion culling calculations.

The normal cone represents the extents of all normal vectors and is therefore sensitive to single erratic vectors that may exist. The normal vector is the centroid of the extents and is therefore not a particularly accurate approximation. A particularly bad case is the representation of a cylinder, where it will be treated as spherical. Non radial solid angles such as the double angled frustum would increase accuracy, but is left for future inclusion.

FIGURE 35. Normal cone with normal N and angle i as (a) acute and (b) obtuse



A parent scene node's normal cone must be calculated from those of the two child nodes only, for speed and independence from leaf scene nodes. Given each child's normal cone vector N with semi angle i, if the two child cones are separate or intersecting and therefore no containership between the two child cones, the following holds, assuming N_A and N_B are normalized:

$$\operatorname{acos}(N_A \cdot N_B) + i_B > i_A$$

i.e. child A does not contain child B and, conversely, B does not contain A:

$$\operatorname{acos}(N_A \cdot N_B) + i_A > i_B$$

then the parent's cone semi angle is given by:

$$i_C = \frac{\operatorname{acos}(N_A \cdot N_B) + i_A + i_B}{2} \qquad \text{(EQ 11)}$$

with normal vector N_C given by:

$$N_C = \left[R_A\right] N_A \qquad \text{(EQ 12)}$$

where $\left[R_A\right]$ is a rotation matrix about the axis formed by the cross product $N_A \times N_B$ that rotates N_A to the new N_C through angle $i_C - r_A$.

5.1.3 Surface Area

When combining attributes such as colours from two child sources to a single parent as an approximation, it would be an incorrect assumption that the two sub-trees should have equal influence. For example, though they may have similar sizes, one may contain far more original surface detail than the other. Balanced trees can not be guaranteed or even assumed in systems where dynamics may occur. In occlusion culling evaluations, it is also necessary to determine the strength of an occluder, which embodies knowledge of the amount of scene detail that may occlude.

Therefore, some form of estimation of the amount of information in each sub-tree is required. A simple count of the number of primitives in each sub-tree could be used, but is of little use when surface samples may be irregular within a single sampling and arbitrarily different between object or scene samplings if combined.

A summated surface area metric appears to be a reasonable choice. Knowledge of the total surface area in each sub-tree allows for a simple weighting scheme when approximating certain attributes in a the parent node.

Given each child node's area w, the parent's surface area is simply:

$$w_C = w_A + w_B \qquad \text{(EQ 13)}$$

Care needs to be taken when sampling designed or procedurally generated surfaces and scenes, to ensure that as little error as possible is introduced when assessing the surface area of a sample scene node. Where possible, area should be measured from the descriptions of the surface region that is represented by a sample, rather than those of the geometry of the sample itself. This also has the added benefit that overlapping regions are not counted twice.

5.1.4 Diffuse Colour

Local illumination models are primarily coloured by diffuse reflection. The commonly used specular and ambient terms will be left as grey unless specified otherwise by other nodes in the scene graph.

Unique colour in each scene node allows subtle texture detail to be included without the need for texture mapping. Also, pre-computed shading can be stored, without the need for one to one texture parameterization for texture baking. An RGBA colour space model will be used, representing Red, Green, Blue and Alpha transparency components.

Given a diffuse colour D for each child node A and B, and child surface areas w_A and w_B, the parent's colour D_C is calculated as a surface area weighted sum of the diffuse colours of the two child nodes:

$$D_C = \frac{w_A D_A + w_B D_B}{w_A + w_B} \quad \text{(EQ 14)}$$

This forms a low pass filter in object space for anti-aliasing.

5.1.5 Attribute Transformations

Scene node attributes must be subject to affine transformations to enable scene graph instancing and dynamics. Instancing is particularly useful when conserving memory by reusing scene components multiple times. Instancing requires a position, orientation and scale to be specified to contextualize the instantiated object in space.

This section will summarize which aspects of affine translation, rotation and scaling apply to each of the attributes in the scene node. Shearing and non uniform scaling has been discarded as it is difficult to integrate into the framework, but little versatility is lost.

TABLE 1. Affine transforms affecting scene node attributes

	Translation	Rotation	Scale
Position	Y	Y	Y
Radius			Y
Normal Cone		Y	
Surface Area			Y
Diffuse Colour			

Diffuse colour space transformations can also be applied for tinting (see Section 5.11.2).

5.2 Scene Sampling

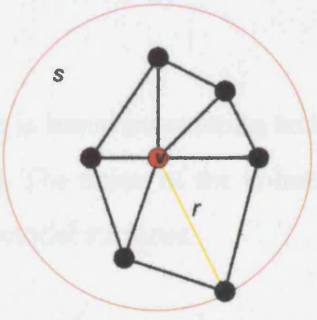
This section details the scene sampling system, introduced in Section 3.4.3. The majority of models used in the graphics and virtual reality field are designed or scanned polygon models. This section will look at two techniques we use to sample polygon scenes, namely vertex patch sampling and voxelization.

Environments must reflect a design. This design may be modelled scenes, procedural methods, or more likely, some hybrid of both, resulting in a procedural system that acts upon modelled scene components, e.g. for 3D texturing. It is important that surfaces are contiguous, without holes. The sampling method must sample not just the geometry of the model, but also the local material properties, transferring them to the scene nodes in a format that will be approximated hierarchically. The nature of the sampling, e.g. the polygon connectivity or known distribution of vertices might be capable of contributing to a hierarchical construction method. However, we generate an unstructured set of samples for generality.

5.2.1 Vertex Patch Sampling

The QSplat system by Rusinkiewicz and Levoy [171] [172] uses a polygon mesh sampling method which guarantees that the volume of the sample set contains all surfaces in the model. A sample sphere s is centred at each mesh vertex. Its radius r is calculated as the maximum length of all triangle edges defined by the vertex, shown in Figure 36.

FIGURE 36. **Triangle patch surrounding vertex V**



This method is suitable for very high resolution meshes but is not applicable to those with larger polygons as over large samples would result. Our system employs this method, but a more versatile voxelization system is normally used.

5.2.2 Voxelization

Voxelization methods have been developed for 3D surfaces in the volume rendering field for primitives such as 3D lines and circles, polygons, general polyhedra and quadric surfaces [118].

Initial work concentrated on aspects of scan line conversion of primitives such as polygons [116] with later work concentrating on anti-aliasing issues and correctness for practical use, often using non scan line based algorithms [47] [105].

Voxelization processes are generally considered to be either binary or non-binary. In the binary case, voxels are considered to be opaque or transparent. Non binary methods are used to anti-alias sampling and consider voxel occupancy to be specified as opacity over the range $[0...1]$. This form of partial occupancy is not used here, but may prove useful in future for anti-aliasing.

Cubic or rectangular voxels can be adjacent either through faces, edges or corners. Some adjacency configurations can cause problems in some volume rendering tasks because traversals are able to pass through what should be a contiguous surface. For example, consider a voxelization of the 3D plane intersecting the origin's X-Y plane as $y = x$. All voxels connect through edges, such that a second plane, intersecting the X-Y plane at $y = -x$, is able to penetrate without the two discrete samplings intersecting. Robustness issues can also occur when detecting a ray's intersection with the edge, such that numerical errors may not detect either case. These issues are inconsequential in the case of this rendering system as voxels will be rendered from object space to image space where coverage is guaranteed.

The sampling method used here is based on existing techniques for the scan line voxelization of polygon models [116]. The union of the volumes of the samples is guaranteed to contain the original polygon model surfaces.

A scan line method has been chosen for several reasons. Firstly, it is efficient, not being greatly more complex than standard polygon rendering. Secondly, it provides uniform surface sample sizes, independent of polygon sizes, that may result in more uniform hierarchies. It is not intended to be a method of generating large numbers of samples for sin-

gle polygons. Though possible, this eventually results in a rendering system that is less efficient than standard polygon rasterization in 2D. Rather, it is intended to sample very high resolution models with just a small number of samples per polygon.

The technique generates samples and stores them in a set and does not need a potentially expensive cubic voxel space data structure. It is also likely to be faster than ray casting methods such as that of the Surfels method by Pfister and colleagues [158] as it does not require ray intersection tests.

A VRML97 rendering engine has been developed that enables scenes to be designed or edited in popular modelling packages such as 3D Studio Max[1], Maya[2] and imported to the system for sampling. A scene may consist of many discrete objects. Therefore, a named scene graph is specified for voxelization. If the set of objects to be sampled does not occur naturally in the same scene, a named VRML97 group node can be added that instances all required scene graphs. This group node can then be specified as the scene graph to sample.

Mean vertex surface normals are automatically calculated for all mesh models with thresholding for discontinuities. Colour per vertex can also be specified per polygon vertex index to enable colour discontinuities.

Given a triangle, the algorithm tests to see which axis the polygon normal is closest to. Information about this orientation is then used to map the triangle's vertex position and normal parameters to a canonical form where the triangle normal is rotated to be closest to the z axis where the triangle is rasterized. The position and geometric attributes are then re-mapped back to the original orientation.

In the canonical position, the triangle's edges are traced and filled along with a given set of attributes for each triangle corner. The canonical z position is interpolated and mapped back to the real orientation. Multiple samples can not exist along the z direction in the canonical space. Care is also taken when developing the algorithm to ensure that multi-

1. (c) Autodesk
2. (c) Autodesk / Alias

ple samples within the same triangle are never generated, to improve efficiency and robustness.

The triangle is voxelized in a uniform voxel space shared by all models undergoing sampling, at a specified density ρ. Each voxel is represented by a scene node with the same position and a spherical radius defined by the voxel's diagonal distance from the centre to a corner that conservatively bounds the voxel's volume:

$$r = \frac{\sqrt{3}}{2\rho} \quad \text{(EQ 15)}$$

It is practical for the voxelization density ρ to be chosen automatically if required, based on the properties of the model. Even if a scene is normalized to a unit volume, the sampling density can be chosen to make use of all detail available. The provided model scale should be preserved in the final result to enable scenes to be composed of collections of objects with correct dimensions.

To sufficiently sample the model, a uniform sampling density ρ can be specified or automatically calculated, based on the distribution of polygon edge lengths in the source scene. Too low a sampling density and potentially available detail may be lost. Too high a density and the object may be over sampled for typical use and require substantially increased storage overhead. A sufficient base distance is chosen as $l = \mu - k\sigma$, equal to the mean μ minus k standard deviations σ where $k > 1$. Chebyshev's theorem [175] states that the minimum proportion of a dataset that lies within k standard deviations of the mean, where $k > 1$, as $1 - (1/k^2)$. When used to establish the base length l, this is a higher proportion of lengths, due to the inclusion of all edge lengths greater than the mean. This places a lower bound on the proportion of surface polygon edges that will receive γ samples. For edge length distributions where this sampling distance estimation l is too small, we use $Max(l, \beta)$ to clip to a minimum threshold distance β.

The sampling density is then calculated using an additional value γ to state the number of samples to be distributed over the base length:

$$\rho = \frac{\gamma}{Max(\mu - k\sigma, \beta)} \quad \text{(EQ 16)}$$

A set of attributes is defined at each triangle vertex, each of which are interpolated along each triangle edge and then across fills. Interpolated attributes include the z position, colour per vertex (if specified), vertex texture and vertex normal vectors.

Texture sampling is calculated simply by interpolating pre-mapped (u,v) within the algorithm. Perspective correction issues that would normally be required in 2D scan line conversion are irrelevant in the 3D case. Aliasing issues can arise when taking a discrete sampling of a texture in this way. Filtering to achieve anti-aliasing is considered in existing methods such as that of Pfister and colleagues [158], but will not be considered here. The (R,G,B,A) colour of 32-bit textures can be sampled, with alpha applied to voxels. Samples with complete alpha transparency are removed. This action is useful for polygons with shapes defined by alpha, e.g. polygons portraying leaf shapes on trees.

A model's conversion to a scene tree in the system is executed at run-time as a procedural generation process, rather than a pre-process. Although current hardware resources are currently not capable of performing the full conversion and scene tree structuring in realtime, the event fits well as a specific type of procedural generator node within the scene description which will be termed a *conversion generator*. The resulting scene graph of a conversion generator can be stored for subsequent use in a binary scene graph format and read on demand.

5.2.3 Surface Area Approximation

The surface area of the sample must be measured during the voxelization process. It is intended to represent the area of the sampled object and not a measurement of the sample's geometry. It is likely that most sampling schemes will have overlaps in the scene nodes used, so that the surface is contiguous. If a surface area measurement based on the samples is used, such as a disc, attempts should be made to subtract area in overlaps such that it is only counted once.

An better measurement results from calculating the surface area of the polygon fragment clipped to the voxel. This gives particularly good evaluations along edges for occlusion culling purposes.

However, for fast sampling and ease of implementation, the method currently used assumes that all voxels have a uniform mean area, calculated by dividing the area of the triangle by the number of samples:

$$a = \frac{|A \times B|}{2n} = \frac{|A||B|\sin\theta}{2n} \qquad \text{(EQ 17)}$$

where A and B are two un-normalized connected edge vectors forming angle θ and n is the number of samples.

5.3 Scene Tree Construction

This section details how a scene tree is constructed, given an unstructured set of scene nodes resulting from polygonal model sampling or any other method that guarantees the definition of contiguous surfaces. See Section 3.4.4 for a basic overview.

5.3.1 Construction Stages

Given a set of scene nodes S, a partitioning stage passes in a top-down manner, to define the binary scene tree structure, set up a hierarchy of containers for tree structure compression (see Section 5.9.2) and calculate hierarchical attributes. In a second top down pass, the uncompressed hierarchy is packed into the container hierarchy created in the first pass, with hierarchical delta encoded geometry compression. These services are made available to all scene graph generator nodes, including standard nodes such as the *group* node.

5.3.2 Pass 1: BSP-Tree Partitioning and Attribute Approximation

Given a set S, an axis aligned bounding box B is calculated. A partitioning plane is placed at the center, bisecting the largest of the three dimensions of B. The scene nodes in S are sorted into two child sets Q_0 or Q_1 such that $Q_0 \cap Q_1 = \{\}$, based on whether each scene node's center is located in front or behind the partition plane. This medial axis technique is used to attempt to minimize the volume of partitioned groups for more efficient level of detail approximations.

An implicit partition plane equation is given by:

$$PV + d = 0 \qquad \text{(EQ 18)}$$

where P is the plane's normal vector and V is any point in world coordinates and d is the plane constant. The normal vector P is that of the chosen partition axis. The constant d is simply calculated by substituting a sample point on the plane in V.

The set S is then partitioned by establishing whether the position of each $V_n \in S$ is in front or behind the partition plane using a simple shortest distance metric to the plane:

$$l = PV_n + d \quad \text{(EQ 19)}$$

where l is the signed distance to the plane. If positive, the node is in front. If negative, it is behind.

At each partitioning, a new branching node is created that represents the nodes being partitioned. Once the tree's structure is identified, its hierarchical attributes must be calculated as functions of the leaf nodes.

Scene nodes are packed into container nodes that represent sub-trees without internal links, for increased storage efficiency. A greedy scene node container packing algorithm reduces the number of required tree pointers by consuming as many tree levels as possible, whilst the sub-tree is still full at height h (see Section 5.9.2). The container then implies required child and parent relationships between scene nodes in the container. Containers are then constructed for each remaining branching node in level h. Special leaf containers are created in cases where all bottom level nodes in the container are leaves and therefore do not require pointers. Containers created in this pass are linked as required.

A simplified recursive C++ function is shown in Figure 37 for the first pass. This function partitions top down, creating a table of uncompressed scene nodes for each tree level, termed the *raw tree*. Pointers are not required in this representation, as the tree's structure is implicitly embodied in the recursive state of the function. Hierarchical attributes are calculated in bottom up order at the end of the function. The required greedy container hierarchy is also created, but their scene nodes are filled in the second pass.

5.3.3 Pass 2: Geometrically Compressed Encoding

Once the first pass has created a raw tree data structure with uncompressed scene nodes in a table and a greedy packing order container hierarchy, the raw tree is then processed in top down, right to left order and packed into the containers, with compression. Delta compression of position and radius requires a top down ordering to encode child node pairs relative to their parent. Scene node normals are stored using a quantization system with quantized cone semi-angle (see Section 5.9.1.2). Colour is stored in a 16-bit 5:6:5

RGB, or 5:6:5:4 RGBA format. To maintain accuracy, surface area is stored as a single precision floating point value.

A simplified recursive C++ function for the second pass is shown in Figure 38. This function encodes two child nodes based on a given parent, given in an uncompressed *Scene_Node_Basic* form. Child nodes are encoded relative to the decoded encoding of the parent node to prevent propagation of errors down the hierarchy. This technique is also used in QSplat [171] [172].

This function is a basic example of an abstract scene graph traversal system that allows client functions to traverse the scene tree transparently of the effects of the scene graph (see Section 5.10.2).

5.3.4 Merging Samples

It is necessary to merge samples when two or more exist at the same or very close proximity positions. These can result from a sampling scheme that produces multiple samples, for example along edges. The simplest course of action is to simply discard duplicates. Duplicate scene nodes are detected during scene tree construction when a set to be partitioned has extents that fall below a tolerance value in all components, but there is more than one item in the set. In this case, either one of the scene nodes can be kept and the rest discarded, or all present can be welded, merging their properties. Many but not all close proximity samples will be grouped, because those that are close on either side of a partition are never considered in the same set.

If welding is used, attributes can be averaged using the operations shown in Table 2.

TABLE 2. Scene node welding methods for duplicate scene nodes

Scene Node Attribute	Weld Method
Position	Arithmetic mean position
Radius	Maximum radius
Normal Cone	Average normals or Merge cones using hierarchical method
Surface Area	Arithmetic mean
Colour	Arithmetic mean colour OR Merge using hierarchical method

FIGURE 37. Scene tree construction pass 1 C++ pseudo code: Partitioning

```
Scene_Node_Control *Scene_Tree_Constructor_KD::construct_optimized_recursive_pass_1(Params &p, unsigned int height)
{
        Scene_Node_Array            nodes_a;
        Scene_Node_Array            nodes_b;
        unsigned short              t, s;
        Scene_Node_Array     *      source;
        Scene_Node_Container *      container;

                                                    // If given node can't be partitioned, exit with NULL
        if(p.node->is_leaf_node())
                return NULL;
                                                    // Set source & dest buffer pointers
        source              = &nodes_a;
        p.dest              = &nodes_b;
        p.container_height  = 1;
                                                    // If node can be partitioned, partition once to source buffer
        p.number_of_leaves = partition_nodes(*p.node, nodes_a[0], nodes_a[1]);

                                                    // Add result to pack from source buffer
                                                    // Partition until at least one node doesn't divide

        while(p.number_of_leaves == 0 && p.container_height < Node::get_max_container_height())
        {
                                                    // Partition one level from left to right
                p.number_of_leaves = partition_nodes_once(*source, *p.dest);

                                                    // Delete all nodes in source
                source->delete_all();
                                                    // Swap source & dest buffers ready for next iteration
                swap(source, p.dest);
                                                    // Increment total tree height counter
                height++;
                p.container_height++;
        }
                                                    // Create leaf or branching container with correct height
                                                    // for local height sub tree
        container = create_container(p);

        s = source->get_number_of_items();

                                                    // For all nodes in last buffer
        for(t = 0; t < s; t++)
        {
                                                    // Get node
                p.node = &(*source)[t];
                                                    // If node can be partitioned
                if(p.node->is_leaf_node() == false)
                {
                                                    // Construct child tree. Node will be added to raw_tree
                                                    // by the function.
                        container->set_external_child(t, construct_optimized_recursive_pass_1(p, height + 1));
                }
                else
                {
                                                    // Bottom node is leaf
                                                    // Set container's external child to NULL
                        container->set_external_child(t, NULL);
                                                    // Add leaf node to raw tree
                        add_node_to_raw_tree(p, height);
                }
        }
                                                    // Delete all items in source tree nodes
        source->delete_all();
                                                    // Refresh container height value after recursion
        p.container_height = container->get_height();
                                                    // Generate all parent Scene_Node_Basic nodes in the container
                                                    // up to a single node root, external to container
        p.k = height - p.container_height + 1;

        p.h = p.container_height;
                                                    // Calculate hierarchical attributes in bottom up order
                                                    // in the raw tree
        for(t = height; t >= p.k; t--)
        {
                p.source_end   = p.raw_tree[t].get_number_of_items();
                p.source_start  = p.source_end - container->get_number_of_nodes_at_height(p.h);

                for(p.i = p.source_start; p.i < p.source_end; p.i += 2)
                {
                        if((p.node_raw = &((p.raw_tree[t - 1]).push_empty())) != NULL)
                        {
                                p.node_raw->calculate_parent(p.raw_tree[t][p.i], p.raw_tree[t][p.i + 1]);
                        }
                }

                p.h--;
        }
                                                    // Return container
        return container;
}
```

FIGURE 38. Scene tree construction pass 2 C++ pseudo code: Encoding

```
void Scene_Tree_Constructor_KD::construct_optimized_recursive_pass_2(Params &p, unsigned int height, Scene_Node_Specifier &parent_spec,
                                                                     Scene_Node_Basic &parent)
{
        Scene_Node_Basic     child;
        Scene_Node_Specifier child_a_spec, child_b_spec;

                                                                     // If leaf, just exit
        parent_spec.get_children(get_scene_tree_manager(), parent, child_a_spec, child_b_spec);
        if(child_a_spec.is_null())
                return;
                                                                     // If no parent specified, return
        if(p.raw_tree[height].get_number_of_items() == 0)
                return;
                                                                     // Get child node specifiers
                                                                     // Get raw node b as Scene_Node_Basic
        child.set(p.raw_tree[height].get_last());
        p.raw_tree[height].pop();
                                                                     // Encode child b
        child_b_spec.encode_node(child, &parent, p.changed);
                                                                     // Get raw node a as Scene_Node_Basic
        child.set(p.raw_tree[height].get_last());
        p.raw_tree[height].pop();
                                                                     // Encode child a
        child_a_spec.encode_node(child, &parent, p.changed);
                                                                     // Decode child for relative encoding of it's tree
        child_b_spec.decode_node(child, &parent);
                                                                     // Construct child tree b
        construct_optimized_recursive_pass_2(p, height + 1, child_b_spec, child);

                                                                     // Decode child for relative encoding of it's tree
        child_a_spec.decode_node(child, &parent);
                                                                     // Construct child tree a
        construct_optimized_recursive_pass_2(p, height + 1, child_a_spec, child);
}
```

5.4 Rendering Functions

The rendering algorithm consists of four stages (see Section 1.7 and Section 3.3). The following sections will examine these stages in more detail, with C++ style pseudo code. The version given here is a non-pipelined version, so each stage executes sequentially.

Additional pseudo code is given for image node refinement and image relation calculation and classification are also given in Section 5.5.4 and Section 5.5.5.

5.4.1 Main Rendering Function

The main rendering function is responsible for invoking each of the rendering stages. Simplified C++ style pseudo code is given in Figure 39. This function simply invokes each stage. All refinement operations act on the image tree data structure, which hides scene graph and scene tree data structures.

The first two stages are invoked, with the third only invoked if the refinement is for a view context and not a light source when creating a shadow map image tree.

Each stage has the option of shadow tracing with one or more light source shadow map image trees held in the *image manager*. Three source/destination arrays are used for image nodes between stages.

5.4.2 Render Refinement Stage 1

The first stage optionally refines the image graph in image tree depth first order, to a level of detail where occludees can be more reliably judged to be occluded. At image scales above threshold t_1, any erroneously occluded image nodes would have substantial impact, so to minimize any chance, the image graph can be refined to a higher resolution. This also commonly alleviates the majority of situations where the viewpoint is contained within scene nodes undergoing refinement. Simplified C++ style pseudo code is given in Figure 40 for stage 1. The first launch function *render_refinement_stage_1()* invokes a recursive function. The function *render_refinement_stage_1_terminate()* decides whether refinement should continue or terminate, based on rendering parameter t_1 (not shown).

5.4.3 Render Refinement Stage 2

The second refinement stage refines the image graph inherited from the first stage, up to the occlusion mask resolution in front to back order, whilst occlusion culling during refinement against image nodes already in the occlusion mask.

Simplified C++ style pseudo code is given in Figure 41. The main launch function *render_refinement_stage_2()* takes nodes resulting from the first stage and filters all image nodes that have zero dependencies in the image graph (see Section 5.5) forming the initial list of order invariant image nodes that will commence refinement in this stage. Nodes in the invariant set U are removed and refined, with child nodes added to U if they have no dependencies. When no nodes exist that are refinable and have no dependencies, this stage terminates.

The second function *calculate_refinement_stage_2()* is given a node to potentially refine. The potential for the node to be an occluder is assessed in *calculate_occluder()* where the node may be committed to the image mask rather than being refined. Otherwise, the node is potentially refined to its children. If the node is a leaf, it is committed to the occlusion mask. Lower level functions handle further additions to the occlusion mask, image graph updates and culling (See Section 5.5.4 and Section 5.5.5).

Note that this version of the algorithm has sequential stages to investigate the algorithm independently of supporting hardware and is not pipelined (see Section 3.10).

FIGURE 39. Main rendering function pseudo code

```
void Render_Manager::render_refinement(Image_Manager &image_manager, System_View_Context &context, bool shadow_trace)
{
                                                                    // Stage 1 refines to threshold level to image node array 1
        render_refinement_stage_1(image_manager, context, shadow_trace, *array[1]);
                                                                    // Pass 2 refines to occlusion mask to image node array 2
        render_refinement_stage_2(image_manager, context, shadow_trace, *array[1], *array[2], *array[3]);
                                                                    // If rendering to display, not a point light source refinement
                                                                    // then render scene nodes
        if(context.is_null() == false)
        {
                                                                    // Pass 3 refine to rasterization resolution to image node array 3
                render_refinement_stage_3(image_manager, context, *array[3], *array[4]);
                                                                    // Render image nodes
                render_image_nodes(image_manager, *array[3], shadow_trace);
        }
}
```

FIGURE 40. Render refinement stage 1 launch and recursive functions

```
void Render_Manager::render_refinement_stage_1(Image_Manager &image_manager, System_View_Context &context, bool shadow_trace Array &dest)
{
        Image_Node * image_node;

                                                                    // Get scene tree root's image node
        if((image_node = image_manager.get_root_image_node(scene_tree_manager)) != NULL)
        {
                                                                    // Initialize root node with shadow traces if enabled
                initialize_image_tree_root(scene_tree_manager, image_manager, *image_node, shadow_trace);
                                                                    // Apply stage 1 to root node
                render_refinement_stage_1_recursive(image_manager, *image_manager.get_camera()->get_position(), context, image_node, dest);
        }

                                                                    // Return OK
        return NULL;
}


void Render_Manager::render_refinement_stage_1_recursive(Image_Manager &image_manager, Vector_3D &viewpoint,
                                                         System_View_Context &context, Image_Node *image_node, Array &dest)
{
        Image_Node              *child_a, *child_b;

                                                                    // Refine nodes to child nodes if possible
        if(refine_image_node(image_manager, *image_node, &child_a, &child_b) == false)
        {
                image_manager.get_image_node_array(Image_Manager::image_node_array_1)->push(image_node);
                return;
        }

                                                                    // Get child a's refinement status
        switch(render_refinement_stage_1_terminate(*child_a))
        {
                                                                    // Image node to be refined
                case refinement_status_refine:
                        render_refinement_stage_1_recursive(image_manager, viewpoint, context, child_a);
                        break;
                                                                    // Image node is terminated in stage 1
                case refinement_status_terminate:
                                                                    // If node has no dependencies, give result to next stage
                        dest.push(child_a);
                        break;
                                                                    // Must be culled
                default: ;
        }

                                                                    // Get child b's refinement status
        switch(render_refinement_stage_1_terminate(*child_b))
        {
                                                                    // Image node to be refined
                case refinement_status_refine:
                        render_refinement_stage_1_recursive(image_manager, viewpoint, context, child_b);
                        break;
                                                                    // Image node is terminated in stage 1
                case refinement_status_terminate:
                                                                    // If node has no dependencies, give result to next stage
                        dest.push(child_b);
                        break;
                                                                    // Must be culled
                default: ;
        }
}
```

FIGURE 41. Render refinement stage 2 launch and refinement functions

```
void Render_Manager::render_refinement_stage_2(Image_Manager &image_manager, System_View_Context &context, Array &nodes,
                                               Array &independent, Array &mask)
{
        Image_Node * node;
                                                                    // For each given node resulting from stage 1
        while(nodes.is_not_empty())
        {                                                           // Remove from stage 1 array
                node = nodes.get_pop();
                                                                    // If node has no dependencies in image graph
                if(node->is_not_dependant())
                        independent.push(node);                     // then add to array of order invariant 'front' image nodes
        }
                                                                    // While valid LOD, order invariant image nodes still exist
        while(independent.is_not_empty())
        {
                node = independent.get_pop();                       // Get last item and pop from stack
                                                                    // If node is not dependant and is not occlusion culled
                if(node->is_not_dependant() && node->is_occlusion_culled() == false)
                {                                                   // Calculate refinement of independent node
                        calculate_refinement_stage_2(image_manager, *node, independent, mask);
                }
        }
}

void Render_Manager::calculate_refinement_stage_2(Image_Manager &image_manager, Image_Node &node, Array &independent, Array &mask)
{
        Image_Node    *child_a, *child_b;

        if(calculate_occluder(image_manager, node))                 // Calculate all affects of possibly being a committed occluder
                return;                                             // and exit if committed occluder

                                                                    // Refine to child nodes based on stage_2 requirements
        if(refine_image_node_stage_2(image_manager, node, &child_a, &child_b, independent) == false)
                commit_occluder(image_manager, node, mask);         // If refinement data not available, commit occluder
}
```

5.4.4 Render Refinement Stage 3

The final refinement stage takes image nodes in the occlusion mask resulting from stage 2 and refines them to rasterization resolutions.

Simplified C++ pseudo code is given in Figure 42. The first launch function *render_refinement_stage_3()* uses the recursive refinement function. The recursive function *render_refinement_stage_3_recursive()* recursively refines to rasterization resolution, decided by the function *render_refinement_stage_3_terminate()*.

The image nodes in the final destination set are rasterized. Note that this version of the algorithm processes sequentially and is not pipelined with the GPU, where passing primitives through the graphics pipeline more immediately, would result in greater performance due to parallel actions of the CPU and GPU.

5.4.5 Render Rasterization Stage 4

The image nodes resulting from the refinement in stage three are sent for rasterization in the fourth and final rendering stage (see Section 5.8).

FIGURE 42. Render refinement stage 3 launch and recursive functions

```
void Render_Manager::render_refinement_stage_3(Image_Manager &image_manager, System_View_Context &context, Array &source, Array &dest)
{
	Image_Node		* node;
	Vector_3D		* viewpoint;

	viewpoint = image_manager.get_camera()->get_position();

	while(source.is_not_empty())				// While occlusion graph image nodes haven't been refined
	{
		node = source.get_pop();			// Get image node and pop
								// Recursively refine image node to rasterization resolution
		render_refinement_stage_3_recursive(image_manager, *viewpoint, context, node, dest);
	}
}


void Render_Manager::render_refinement_stage_3_recursive(Image_Manager &image_manager, Vector_3D &viewpoint, System_View_Context &context,
							Image_Node *image_node, Array &dest)
{
	Image_Node		*child_a, *child_b;
								// Refine nodes to child nodes if possible
	if(refine_image_node_stage_3(image_manager, *image_node, &child_a, &child_b) == false)
	{
		dest.push(image_node);			// If not available, add to rasterization set and exit
		return;
	}
								// Get child a's refinement status
	switch(render_refinement_stage_3_terminate(*child_a))
	{
								// Image node to be refined
		case refinement_status_refine:
			render_refinement_stage_3_recursive(image_manager, viewpoint, context, child_a, dest);
			break;
								// Image node is terminated in stage 1
		case refinement_status_terminate:
								// If node has no dependencies, give result to next stage
			dest.push(child_a);
			break;
								// Must be culled
		default: ;
	}


	switch(render_refinement_stage_3_terminate(*child_b))
	{
								// Image node to be refined
		case refinement_status_refine:
			render_refinement_stage_3_recursive(image_manager, viewpoint, context, child_b, dest);
			break;
								// Image node is terminated in stage 1
		case refinement_status_terminate:
								// If node has no dependencies, give result to next stage
			dest.push(child_b);
			break;
								// Must be culled
		default: ;
	}

}
```

5.5 Image Graph Refinement

This section first describes the image graph data structure's image nodes and image relations and then details how the image graph is refined during rendering. See Section 1.7 and Section 3.3 for an introduction on the rendering stages and image graph refinement.

5.5.1 Image Node

The image node is an image space counterpart of the object space scene node. Scene tree data represents view independent information about the geometry of the scene. Viewpoint dependent information and other rendering state data are encapsulated within the image node. Scene graph instancing causes more than one image node to be created for a scene node as each are used in a different context and therefore have different viewpoint dependent properties. The attributes associated with each image node are listed in Table 3 with brief descriptions and will be discussed in later sections.

An image node is associated with one scene node, which it stores in a decompressed form. It also requires a scene node specifier described in Section 5.9.2.3 that provides access to the scene tree node's parent and child nodes using an abstract traversal that shields the rendering algorithm from the underlying details of the scene graph.

Viewpoint dependent projection information is also stored. The projection of the sphere is encapsulated by a view cone angle with view vector from the scene node to the viewpoint. Other information includes the distance of the scene node from the viewpoint and the view angle between the normal cone vector and the view vector.

Image graph relations must be associated with the node, as is a child link to two child image nodes in an image node equivalent of a 1-container (see Section 5.9.2.1) to form the image tree of the refinement process to support caching.

A set of links to light source image tree nodes that currently illuminate the image node if it is in a viewpoint image tree, is required for image tree shadow tracing (see Chapter 4).

An occlusion value is included to support estimates of aggregate occlusion of the node and an occlusion strength value estimates the image node's suitability as an occluder of other nodes.

A dependency counter is zero or greater and counts the number of relations with nodes that require refinement before the image node itself to enforce a front to back refinement ordering.

Finally, a number of binary flags represent the node's state for reference when the node undergoes any processing.

TABLE 3. Summary description of image node attributes (V = View Dependent)

Image Node Attribute	Description
Scene Node	Decompressed representation of associated scene node.
Scene Node Specifier	Smart pointer to scene node in scene graph container.
View Cone (V)	View of spherical scene node stored as view vector and angle from vector.
View Distance (V)	Length of view vector from viewpoint.
View Angle (V)	Angle between view vector and scene node's normal cone.
Image Graph Relation Set	Set of relations between this node and neighbouring nodes.
Image Tree Link	Links to child image nodes that represent the image tree.
Shadow Trace Link Array	References to illuminating image nodes in light source image trees.
Occlusion Value (V)	Measure of image node's aggregate occlusion.
Occlusion Strength (V)	Measure of image node's effectiveness as an occluder.
Dependency Counter (V)	Number of other image nodes this image node is dependent on to ensure required traversal ordering.
Render State Flags (V)	Binary state flags storing current state of image node during rendering. Flags are included to indicate: 1. View volume plane intersection (6 for frustum) 2. Viewpoint contained 3. View volume culled 4. Backface culled 5. Occlusion culled 6. Committed to occlusion mask 7. Pending refinement

5.5.2 Image Relation

An image relation is an arc in the image graph that represents the 3D and viewpoint dependent 2D relationship between a pair of image nodes and their associated scene nodes. The relation is directed, such that the first is considered the dominant node in the relationship. The 2D image space and 3D object space relation states are summarized in Table 4. The use of these states will become clearer as the refinement process is described.

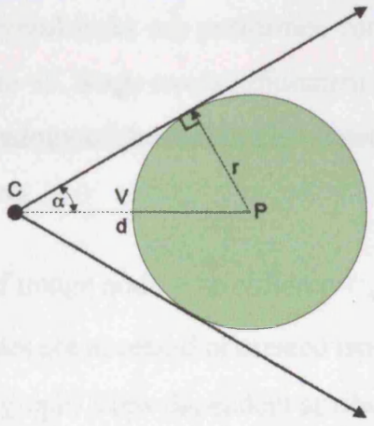
TABLE 4. Mutually exclusive relation states in 2D and 3D

State Type	Mutually Exclusive Relation States
2D	Separate 2D | Intersecting or Contained 2D | Occluder Contained 2D | Null 2D
3D	Separate 3D | Intersecting 3D | Contained 3D

5.5.3 Image Node View Cone and View Angle

A scene tree's scene node is considered to be a sphere in object space. The view of the sphere forms a view cone with its apex at the camera's centre of projection and its sides touching the sphere surface such that it is tangential, shown in Figure 43.

FIGURE 43. Image node view cone angle α from viewpoint C, radius r, distance d



Given a scene node positioned at P with radius r, viewed from centre of projection C, a view vector $V = C - P$ is calculated, with its length (view distance) d.

The view cone half angle α is given by:

$$\alpha = \text{asin}\left(\frac{r}{d}\right) \quad \text{(EQ 20)}$$

The centre of projection C is touching or contained by the scene node if $d \leq r$. This is a special case that is flagged in the image node as *viewpoint contained*, so that no image space operations are attempted on the image node.

The view angle θ is the angle formed between V and the normalized normal cone vector N described in Section 5.1.2 given by:

$$\theta = \text{acos}\left(\left(\frac{V}{d}\right) \cdot N\right) \quad \text{(EQ 21)}$$

5.5.4 Image Node Refinement

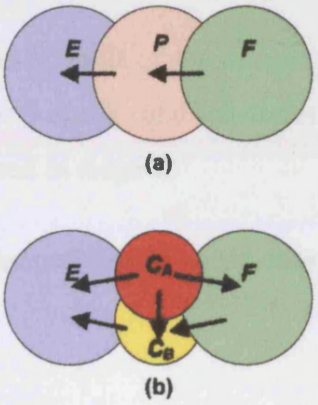
The image graph represents the state of image refinement at a given time. The rendering algorithm can use a refinement ordering indicated by the image graph's dependency counting, or refine the image graph in arbitrary order, where the graph will remain valid.

An image node has a set of relations with other image nodes, declaring 2D and 3D geometrical relationships between them, listed in Section 5.5.2. When a parent image node is refined to its child pair, several tasks are performed for each child, as shown in the C++ like pseudo code in Figure 45. Stage two's refinement is shown, but other stages are similar. This is a simplified analogy of the real implementation which makes use of optimized logic paths and functions.

Figure 44 shows refinement of image node P to children C_A and C_B, with external nodes E and F. The child image nodes are accessed or created using the image tree. The parent P is removed from the image graph. View dependent attributes are calculated for C_A and C_B, such as the view cone and view angle. Back face culling is then carried out as required by the inherited state from the parent, discussed in Section 3.6.2 and Section 5.6.2. At this point, one or both of the child nodes may have been culled, the result flagged in the child image nodes. Based on this result, view volume culling may or may not be carried out, based on the frustum culling state inherited from the parent image

node, discussed in Section 3.6.1 and Section 5.6.1. If neither children are back face or view volume culled, an attempt is made to establish a relation between them, shown in Figure 44(b). Then, if a child is not culled, an attempt is made to establish new relations between it and the external nodes its parents had relations with, shown in Figure 44(b) with E and F. It is then added to the image graph and is scheduled for refinement if its dependency counter is zero. Lastly, the external image nodes that the parent had relations with are tested for schedulability due to a zero dependency counter. See Section 3.3.3 for a detailed example of image graph refinement.

FIGURE 44. Refinement of P to child nodes C_A and C_B with externals E and F



In practice, refinement for each rendering stage is marginally different and can exploit small differences for speedups. Refinement stage two's functions will be given as an example here, because they are the most complex. Simplified C++ pseudo code is shown in Figure 45 for the function *refine_image_node_stage_2()*. It first refines the given node to child nodes if possible and calculates their standard view dependent attributes. Back face and view volume culling are then invoked, all image relations finally being calculated.

Simplified C++ pseudo code for stage two's *calculate_relations_stage_2()* is shown in Figure 46. This function caters for each combination of culling in the child nodes, establishing any required relations between the two child nodes themselves, if they are both not culled and other nodes that their parent image node had relations with. It is also responsible for applying occlusion culling contributions. A relationship between two image nodes is analyzed and potentially established by the function *classify_relation()*, shown in Figure 47 for two nodes.

FIGURE 45. Pseudo C++ code for parent image node refinement to child image nodes

```
bool Render_Manager::refine_image_node_stage_2(Image_Manager &image_manager, Image_Node &image_node,
                                               Image_Node **child_a, Image_Node **child_b)
{
                                                                    // Get child image nodes if they exist
    if(image_manager.get_children(scene_tree_manager, image_node, child_a, child_b) == false)
        return false;

    Vector_3D &viewpoint = *image_manager.get_camera()->get_position();
                                                                    // Calculate solid angles (& view vectors)
                                                                    // of the two new child image nodes
    (*child_a)->calculate_standard_attributes(viewpoint);
    (*child_b)->calculate_standard_attributes(viewpoint);
                                                                    // Calculate child a & b's backface culling state
    backface_cull(**child_a, **child_b);
                                                                    // Calculate child a & b's clipping state,
                                                                    // based on the clipping state of the parent
    view_volume_cull(*image_manager.get_camera(), image_node, **child_a, **child_b);
                                                                    // Calculate all relations for a and b
                                                                    // based on their clipping state
    calculate_relations_stage_2(image_manager, image_node, **child_a, **child_b);
                                                                    // Return child nodes exist
    return true;
}
```

If any of the nodes involved, as a result of the refinement, have no dependencies, the function *calculate_current_layer()* checks and schedules nodes for refinement, to realize the front to back ordering required in stage 2.

Pseudo code for a simplified relation classifying function is given in Section 5.5.5.

FIGURE 46. Pseudo C++ code to calculate relations between image nodes in stage 2

```
void Render_Manager::calculate_relations_stage_2(Image_Manager &image_manager, Image_Node &parent_node, Image_Node &child_a, Image_Node &child_b)
{
	Image_Relation		*	relation;
	Image_Relation		*	previous = NULL;
	Image_Node		*	other_node;
	Image_Graph		*	image_graph = image_manager.get_image_graph();

	if(child_a.is_culled() == false)					// If child_a is not culled
	{
		if(child_b.is_culled() == false)				// If child b is not culled
		{
			classify_relation(*image_graph, child_a, child_b);	// Establish any relation between two child nodes
										// For each relation in the refined parent node
			for(relation = parent_node.get_first_relation(); relation != NULL; relation = relation->get_next(parent_node))
			{
				other_node = relation->get_other_node(parent_node);
										// Establish any relation between static node and two child nodes
				classify_relation(*image_graph, *relation, *other_node, child_a, child_b);
				remove_node_relation(*image_graph, *other_node, *relation);
										// Remove relation from other node's list
				calculate_current_layer(image_manager, *other_node);
										// If other node is in current layer, schedule node
			}
			if(calculate_occlusion_culling(image_manager, child_a) == false)
			{
				calculate_current_layer(image_manager, child_a);
			}
			if(calculate_occlusion_culling(image_manager, child_b) == false)
			{
				if(child_b.is_pending_refinement() == false)
				{
					calculate_current_layer(image_manager, child_b);
				}
			}
		}
		else
		{
			for(relation = parent_node.get_first_relation(); relation != NULL; relation = relation->get_next(parent_node))
			{
				other_node = relation->get_other_node(parent_node);
				classify_relation(*image_graph, *relation, *other_node, child_a);
				remove_node_relation(*image_graph, *other_node, *relation);
				calculate_current_layer(image_manager, *other_node);
			}
			if(calculate_occlusion_culling(image_manager, child_a) == false)
			{
				calculate_current_layer(image_manager, child_a);
			}
		}
	}
	else
	{
		if(child_b.is_culled() == false)
		{
			for(relation = parent_node.get_first_relation(); relation != NULL; relation = relation->get_next(parent_node))
			{
				other_node = relation->get_other_node(parent_node);
				classify_relation(*image_graph, *relation, *relation->get_other_node(parent_node), child_b);
				remove_node_relation(*image_graph, *other_node, *relation);
				calculate_current_layer(image_manager, *other_node);
			}
			if(calculate_occlusion_culling(image_manager, child_b) == false)
			{
				calculate_current_layer(image_manager, child_b);
			}
		}
		else
		{
			for(relation = parent_node.get_first_relation(); relation != NULL; relation = relation->get_next(parent_node))
			{
				other_node = relation->get_other_node(parent_node);
				remove_node_relation(*image_graph, *other_node, *relation);
				calculate_current_layer(image_manager, *other_node);
			}
		}
	}
}
```

5.5.5 Image Relation Classification

The occlusion culling technique overviewed in Section 3.7 assesses aggregate occlusion of image nodes by other image nodes. To facilitate this, the image graph is refined when image nodes are refined, maintaining 2D image based and 3D object space based classifications of relationships between neighbouring nodes. Only geometrical relationships are assessed, with actual occlusion estimations performed when required in rendering stage two (see Section 5.7). The next sections discuss the types of classification possible and how the classifications are established. Optimizations are then examined, using hierarchical coherence.

5.5.5.1 Classifications

The matrix shown in Table 5 shows all relation geometric classifications between a pair of nodes, where *separate, intersecting* and *contained* configurations are of interest in both 2D image space and 3D object space.

If nodes are separate in 2D, they must also be separate in 3D. If this classification arises, no relation is created between the nodes as they are not overlapping in the image in any way and therefore have no depth ordering or occlusion.

If the nodes are separate in 3D, their image node counterparts may be intersecting in 2D or one may contain the other in 2D.

If nodes are intersecting in 3D, their image nodes must also be intersecting in 2D or one may contain the other in 2D.

If one scene node contains the other, there is only one possible relationship in 2D, that the containing scene node is also the containing image node.

A special case occurs during rendering where a scene node may contain the viewpoint in 3D. In this case, no discrete image space projection of the scene node is possible and the 2D classification is Null_2D. During refinement, nodes with these states are given prior-

ity as the first node in the relation, to ensure they are refined first. If both nodes contain the viewpoint, the order is considered arbitrary.

TABLE 5. Image relation classifications possible in 2D and 3D

	Separate 3D	Intersecting 3D	Contained 3D	Viewpoint Contained 3D
Separate 2D	Y			
Intersecting 2D	Y	Y		
Contained 2D	Y	Y	Y	
Null 2D				Y

In practice, a set of classifications derived from this table is used, where 2D intersection and containment are merged to one state *intersecting_or_contained_2d.* A 2D state where the occluder is contained by the occludee is also added as *occluder_contained_2d.*

TABLE 6. Image relation classifications in 2D and 3D altered for hierarchical coherence

	Separate 3D	Intersecting 3D	Contained 3D	Viewpoint Contained 3D
Separate 2D	Y			
Intersecting or contained 2D	Y	Y	Y	
Occluder contained 2D	Y	Y		
Null 2D				Y

These modifications support specific hierarchical coherence optimizations (see Section 5.5.5.4) when classifying a relation between a child and an external node, based on the previous classification between the parent and the external node and are summarized in Table 6.

Simplified C++ pseudo code for a simple classification algorithm is given in Figure 47, for the case of two arbitrary nodes. In practice, multiple functions exist for optimized logic paths and hierarchical coherence optimizations.

FIGURE 47. Pseudo C++ code relation classification between two image nodes

```
void Render_Manager::classify_relation(Image_Graph &image_graph, Image_Node &node_a, Image_Node &node_b)
{
        Angle_Quantized         separation;
        Scalar                  distance_squared;
        Image_Node              * node_front, * node_back;

                                                                        // If either node contains viewpoint, form relation and exit
        if(classify_relation_viewpoint_contained(image_graph, node_a, node_b))
                return;
                                                                        // If image nodes are intersecting or contained in 2D
        if(node_a.is_separate_2d(node_b, separation) == false)
        {
                distance_squared = node_a.get_distance_squared(node_b);         // Calculate square of distance between scene nodes

                if(node_a.is_separate_3d(node_b, distance_squared))             // If image nodes are separate in 3D
                {
                        node_front = node_a.get_nearest(node_b, &node_back);    // Get which node is at front and which is at back
                                                                                // If back node contains front in 2D, set relation
                                                                                // otherwise nodes are intersecting or front is container
                        node_front = node_front->get_occluder(&node_back);      // Get which is occluder in relationship (handle exceptional cases)

                        if(node_back->is_container_2d(*node_front, separation))
                                create_new_relation(image_graph, *node_front, *node_back, occlusion_occluder_contained_2d | occlusion_separate_3d);
                        else
                                create_new_relation(image_graph, *node_front, *node_back, occlusion_intersecting_or_contained_2d | occlusion_separate_3d);
                }
                else
                {
                        if(node_a.is_containment_3d(node_b, distance_squared))  // If image nodes have containment status
                        {
                                node_front = node_a.get_container_3d(node_b, &node_back);
                                                                                // Set relation's image nodes and 3D containment status
                                create_new_relation(image_graph, *node_front, *node_back, occlusion_intersecting_or_contained_2d | occlusion_contained_3d);
                        }
                        else
                        {
                                                                                // Nodes are occlusion_intersecting_3d
                                                                                // Establish which will be considered to be at the front and back
                                node_front = get_intersection_3d_order(node_a, node_b, &node_back);
                                                                                // Nodes are intersecting in 3D, intersecting or contained 2D
                                create_new_relation(image_graph, *node_front, *node_back, occlusion_intersecting_or_contained_2d | occlusion_intersecting_3d);
                        }
                }
        }
}
```

5.5.5.2 3D Relation Classification

The 3D relationship must be efficiently established between two scene nodes. Given two scene nodes A and B, a test for geometric 3D separation is simply given by:

$$|P_A - P_B|^2 > (r_A + r_B)^2 \qquad \text{(EQ 22)}$$

with scene node positions P_A and P_B, with radii r_A and r_B. Note that the requirement for the square root has been removed when calculating the vector length.

If one node, say A contains B, the following holds:

$$|P_A - P_B| < r_A - r_B$$

and vice verse if B contains A. To remove the requirement for the square root when calculating the vector length and make a general containment test, both sides are squared:

$$|P_A - P_B|^2 < (r_A - r_B)^2 \qquad \text{(EQ 23)}$$

Squaring makes the sign of the right hand side indeterminable and therefore serves as a general test of whether one scene node contains the other. If this test is true, the containing and contained are distinguished by the larger and smaller radius respectively.

If the containment test is false, the default case arises where A and B are intersecting.

Note that relation node ordering is usually the nearest as the first to be refined, but cases can occur where far nodes are occluders in separate 3D relations (see Section 3.3.5).

5.5.5.3 2D Relation Classification

The view cone in Section 5.5.3 describes the projection of the sphere towards the centre of projection. The footprint on the image plane is an elliptical conic section.

Tests are required to establish intersection and containment relationships in 2D.

Given two image nodes, no relation need to exist between them or nodes in their child trees if the nodes themselves are geometrically separate in the image. Distinguishing between intersection and occluder containment in 2D also allows hierarchical coherence optimizations.

Given two image nodes A and B, their normalized view cone vectors V_A and V_B, with cone semi-angles α_A and α_B, the two view cones are separate iff:

$$\text{acos}(V_A \cdot V_B) > \alpha_A + \alpha_B \qquad \text{(EQ 24)}$$

If not separate, view cone A contains view cone B, iff:

$$\text{acos}(V_A \cdot V_B) < \alpha_A - \alpha_B \qquad \text{(EQ 25)}$$

or then vice versa, view cone B contains view cone A iff:

$$\text{acos}(V_A \cdot V_B) < -(\alpha_A - \alpha_B) \qquad \text{(EQ 26)}$$

This test can be used to establish whether an occluder in a relationship is contained within the occludee in 2D for logic optimizations. If the view cones are not separate and one does not contain the other, the view cones must be intersecting.

5.5.5.4 Hierarchical Coherence Optimization

Hierarchical coherence can be used to optimize logic paths when classifying a relation between a child and an external node, based on the classification of the parent and the external node.

TABLE 7. Inheritable image relation states from parent to child relation

	Inheritable relation state from parent relation to child relation	Occluder / Container Refined	Occludee / Contained Refined
(a)	Separate_3d	Y	Y
(b)	Contained_3d		Y
(c)	Contained_2d	Y	Y

Given a relation between a parent and an external node, certain classifications can be inherited by the child relation without further calculation. Four possible inheritable states are shown in Table 7, based on whether it is the occluder/container or occludee/contained undergoing refinement. The case most likely to occur is (a), where many pairs may be separate in 3D. Depth ordered refinements will not use case (b), because it will refine the container node before the contained, but this may occur in un-ordered refinement. Case (c) occurs frequently during stage two's front to back refinement, but only usually where the occluder is contained by the occludee in 2D and is undergoing refinement, because farther nodes remain large in image space, while nearer nodes are smaller and undergoing refinement. To distinguish this condition, the *contained_2d* state is split into two separate states *occluder_contained_2d* and *intersecting_or_contained_2d*, where in the latter, containment of the occludee in 2D is not distinguished and is merged with the intersection state (see Section 5.5.5.1).

Care must be taken during implementation, such that the logic paths used to detect coherent states do not themselves outweigh the speedup provided by the optimization. This is particularly of note in hardware architectures where missing branch prediction can incur substantial overheads in the CPU. These optimizations have been compared against a simpler approach of simply forming unclassified relations when image nodes overlap in image space and classifying some relations later, when required. However, this optimized version has shown to run about 15% faster than the simpler late classification.

5.6 View Volume & Back Face Culling

This section discusses the implementation of the hierarchical view volume and back face culling techniques, introduced in Section 3.6.

5.6.1 View Volume Culling

Given a scene node sphere A and a view volume Q, a test is required to establish whether the sphere is outside, intersecting or inside Q. Here, we will assume that the view volume is defined by a set of planes forming a finite convex view volume, such as the frustum (see Section 1.2).

If A is outside Q, it can be culled during rendering. If it is wholly inside Q, it is not culled and no scene nodes in A's child tree need be view volume tested. If A intersects one or more planes $P \in Q$, then A is partially visible and child nodes in A's scene tree must be view volume tested.

An image node I representing A in image space, stores intersection bit flags F, with one bit per plane P that is set if A intersects P or clear if not. This state is inherited by the child image nodes of I. When a child inherits F, if $F = 0$, the parent scene node is wholly in Q and no planes need to be tested. If F is non-zero, then the child is tested against the specific planes indicated by set bits in F.

Before rendering, the view volume is projected into object space, where the spatial tests can be carried out, such that each scene node need not be projected into camera space. If scene node A has position V, its state can easily be found by applying the plane equation:

$$s = VP + d \qquad \text{(EQ 27)}$$

where s is the distance of A from the plane, P is the plane normal vector and d is the plane equation constant. If plane normal vectors in Q point outward from the view volume, then if $s > r$ where r is A's radius, then A is outside P. If $s < -r$, then A is inside P. Otherwise, A intersects P. To class A as outside Q, at least one s value for any $P \in Q$ must be outside. To be classed as inside, all s values for all $P \in Q$ must be inside.

5.6.2 Back Face Culling

A method is needed to establish from the view of a scene node with a normal cone, whether all scene nodes in its child tree are back facing. If there is potential that a node in its child tree may be front facing, the scene node can not be culled. The approach used is similar to the Spatialized Normal Cone by Johnson and Cohen [111]. Figure 48 shows a scene node with view vector and view cone from the viewpoint. The diagram is a cross section in the plane containing the centroid normal vector N and view vector V. The normal cone has semi angle i, bounding the extent of all normals in the node's child tree. The view cone has axis V and semi angle α. The view angle between V and N is θ. The wing angle $w = i + \pi/2$ specifies the maximum angle at which a view vector may see a surface in the normal cone. Part (a) shows the construction centered at the scene node's position. However, leaf scene nodes may be positioned anywhere within the spherical volume with any normal in the normal cone and thus visibility is also a function of their potential positions. The extremity cases are where a child scene node is positioned on the view cone. Part (b) shows how the center of the construct in part (a) is rotated to meet the node's view cone angle. The new view angle $q = \theta + \alpha$. The same case must also be considered for the opposite side of the cone angle, where $p = \theta - \alpha$ (not shown).

FIGURE 48. Hierarchical back face culling geometry

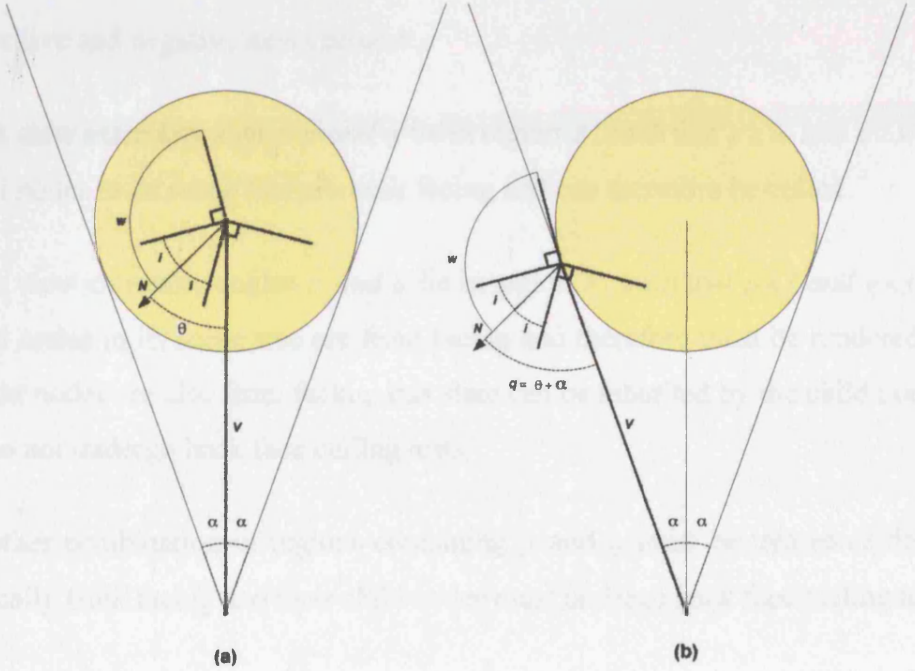
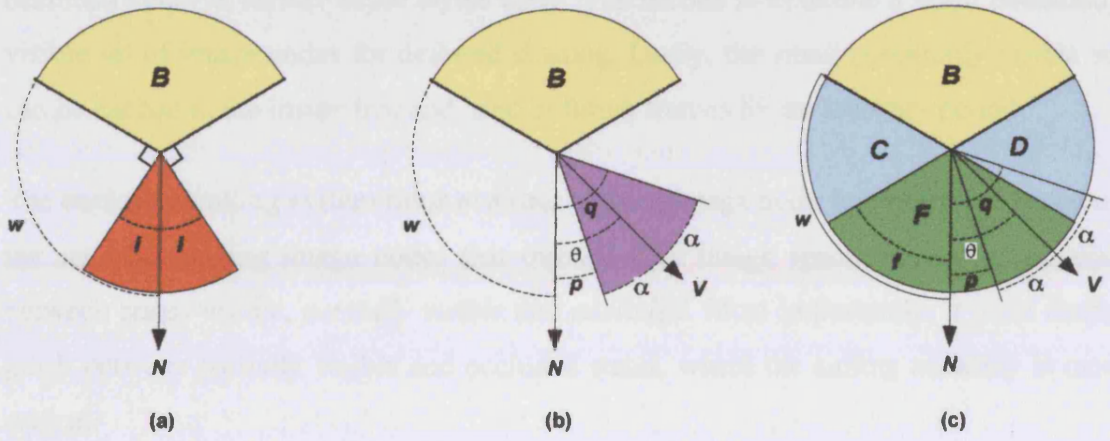


Figure 49(a) shows the scene node normal cone with angle i, defining a back face region B by wing angle $w = \pi/2 + i$. Any view vector in B will view the back of any scene node in the child scene tree. Part (b) shows the extremity view vector cases in relation to the scene node. Two extremity angles $p = \theta - \alpha$ and $q = \theta + \alpha$ denote the limits of all possible view vectors from the viewpoint to the scene node, due to the image node view cone angle α measured linearly as a cone from the view vector V.

FIGURE 49. Scene node normal cone visibility



Part (c) shows mixed front and back regions C and D between angles $f = \pi/2 - i$ from N and w. This mixed region represents the model's silhouette boundaries in the image. The region F defines a front facing region. Note that F and B are the same open cone over the positive and negative axis vector N.

If both view extremity angles p *and* q lie in region B, such that $p \geq w$ *and* $q \geq w$, the node and all nodes in its scene tree are back facing and can therefore be culled.

If both view extremity angles p *and* q lie in region F, such that $p < f$ *and* $q < f$, the node and all nodes in its scene tree are front facing and therefore must be rendered. Because all child nodes are also front facing, this state can be inherited by the child nodes so that they do not undergo back face culling tests.

Any other combination of regions containing p and q must be treated as definitely or potentially front facing and their child nodes must undergo back face culling tests.

5.7 Occlusion Culling

This section discusses implementation details of the occlusion culling process described in Section 3.7, based on image graph refinement (see Section 3.3 and Section 5.5). The reader is referred to Section 3.7, but a short summary is also provided here before detailing each of the occlusion culling system's underlying functions.

Occlusion culling in this system aims to provide three benefits. The first is to remove occluded detail in further depth layers early. The second is to define a small potentially visible set of image nodes for deferred shading. Lastly, the small potentially visible set can be cached in the image tree and used in future frames for an arbitrary period.

The occlusion culling system must evaluate a given image node for occlusion, based on the set of occluding image nodes that overlap it in image space. It must distinguish between states *visible*, *partially visible* and *occluded*. Most importantly, it must distinguish between partially visible and occluded states, where the culling accuracy is most critical.

Occlusion culling relies on the image graph to provide a front to back ordering and give information about which occluders cover an image node. During the front to back ordering, the scene is refined to build an occlusion mask, against which the further image nodes are tested for occlusion later in the image graph's refinement.

The occlusion estimation, although assessing an aggregate coverage, only considers unique occlusion contributions from each occluder, without analysing the effects of occlusion between the occluders for performance reasons. At the outset of this research, it was not known whether a reliable classification can be made using this approach, but results shown in Chapter 6 indicate a reasonable level of reliability.

Several sub-problems are defined. The first is to estimate the *strength* of an occluding image node. That is, how well it is able to occlude. This depends on the amount of scene detail present in the node, that is measured as surface area. It also depends on how this surface area is distributed within the scene node and how much of this surface area is visible to the viewpoint to act as an occluder.

The extent to which the occluding image node covers the occludee image node in image space is then measured as a coverage ratio between the two view cone solid angles.

An occlusion contribution is then made to the occludee's *occlusion value*, based on these measurements. The occludee may receive zero or more occlusion contributions from occluders, which are summated. The culling decision is made by thresholding the occlusion value.

5.7.1 Image Node Occlusion Estimation Functions

Child image nodes that result from the refinement of a parent image node in rendering stage two, undergo occlusion estimation and are potentially culled. Each child may have zero, one or more occluders that are committed to the occlusion mask M against which they undergo occlusion testing and potential culling.

The process of image node occlusion estimation and culling relies on four functions:

1. Occlusion Contribution
2. Occlusion Strength (PVA)
3. Solid Angle Occlusion Ratio
4. Occlusion Estimation (summation)
5. Occlusion Cull (threshold)

The *occlusion contribution* function $OccContrib(A, B)$ produces a value that is added to occludee image node A's *occlusion estimation* value. This estimates the level of occlusion between occludee A and occluder B. Two main factors contribute to the outcome. The first is B's *occluder strength*, given by $OccStrength(B)$ (see Section 5.7.2) in the range $[0...1]$ where 0 is no occlusion and 1 is total occlusion. The second is how much of the occludee A is covered by the occluder B in the perspective projection. The *solid angle occlusion ratio* function $OccRatio(A, B)$ (see Section 5.7.3) calculates how much of A's view cone solid angle is occluded by B's as a clipped value $[0...1]$ where values 0 and 1 represent no occlusion and total occlusion of A respectively. The resulting occlusion contribution in the range $[0...1]$ is the product of the occlusion strength and the occlusion ratio:

$$OccContrib(A, B) = OccStrength(B)OccRatio(A, B) \quad \text{(EQ 28)}$$

The *occluder strength* function $OccStrength(B)$ is applied to all image nodes that are committed to the occlusion mask M in rendering stage two, with the resulting value stored in the image node. This evaluates the node's ability to occlude. The technique developed, called the *Parallel Visible Area* (PVA) function has both view independent and view dependent aspects. It calculates the ratio between the image node's hierarchical surface area (see Section 5.1.3) when distributed over its normal cone, that is estimated to be component to the viewpoint and the sphere's cross section disc area.

Each occlusion contribution counts towards A's aggregate *occlusion estimation* value using a summation function:

$$OccEstimation(A) = \sum_{i=1}^{n} OccContrib(A, O_i) \quad \text{(EQ 29)}$$

where O is the set of n occluding images nodes in the occlusion mask with relations with occludee node A, obtained by the function from A.

The final *occlusion culling* function $OccCull(A, t_{OccCull})$ estimates whether the occludee A is to be culled, using a simple thresholding function based on $t_{OccCull}$:

$$OccCull(A, t_{OccCull}) = \begin{cases} \text{true if } OccEstimation(A) > t_{OccCull} \\ \text{false otherwise} \end{cases} \quad \text{(EQ 30)}$$

where $t_{OccCull}$ is a threshold value chosen empirically. The value is generally a small multiple of an ideal, unit occlusion value, with extra contingency for typical overlap between occluders. The effects of a low value, good value and high value 1, 1.9 and 10.0 are demonstrated in Section 6.4, with 1.9 shown to be an effective choice.

This form of summated contributions with thresholding may produce incorrect results in some situations. For example, when looking along a row of objects at a shallow angle, a large number of contributions may be made, even though the occluders are weak. Enough contributions may be made to trigger culling. However, this function serves as a basis for analysis and future development. Alternative functions may be developed, or additional terms may be introduced to penalize incorrect situations, for example, based on distances between occluder and occludee to help prevent near field self occlusion.

For complex vista style views where many silhouettes may be visible over many depth layers, to prevent erroneous culling at more distant silhouettes, occluders can be discarded at low values by thresholding $OccluderStrength(B) < t_{OccClip}$ such that they do not make any occlusion contributions.

The next sections will discuss the PVA and solid angle occlusion ratio functions used in the occlusion strength and occlusion contribution functions.

5.7.2 Occluder Strength Estimation - The PVA Function

When considering how well a single occluder covers a single occludee, both their relative positions in the image and the surfaces that they contain at higher resolutions will both contribute to the outcome. However, without analysis at higher resolutions, hierarchical descriptions are the only source of information on which to base decisions.

The *occluder strength* is a measure of how well an occluder can occlude potential occludees, regardless of their relative positions in the image. The area of the occludee covered by the occluder is then used in combination with the occluder strength to result in an occlusion contribution value which is made to the occludee.

During the occlusion culling phase of the rendering algorithm, image nodes are committed to the occlusion mask M. The occluder strength can be assessed once at this time and used in all future occlusion calculations with potential occludees.

When considering methods of estimating an occluder strength, the solution may be view dependent or view independent. That is, it may take the particular view of the occluder into account, or simply apply a general estimation for any view. A view dependent technique may be able to account for approximations of visible surface distribution within the scene node for increased accuracy, particularly along object silhouettes. A view independent algorithm would need to generalize for any view of the node, but may be faster.

To limit analysis to the occluder node itself without addressing higher resolution information, data must be stored in the scene node that will support the chosen method. This data must be very small to permit the storage and processing of large numbers of scene nodes.

Viewpoint independent methods for strength estimation were discarded with the expectation of low quality results. However, a totally viewpoint dependent method does not lend a great deal to the algorithm because explicit higher resolution scene nodes present in the occluder's child scene tree are not accessed to attempt to achieve higher performance.

The *Parallel Visible Area* (PVA) function developed here is partially viewpoint dependent. Given a viewpoint and an occluder whose strength will be evaluated, it estimates the amount of surface area that is visible to the viewpoint under a parallel projection. Therefore, only the orientation of the node relative to the viewpoint is considered. The distance of the scene node from the viewpoint is not taken into consideration.

Rather than treat the occluder as a sphere, where occlusion strength is non-uniform over the view cone due to different distances that would be travelled by rays in the cone through the sphere (low at the periphery, high at the center), a simplification is made such that occluder strength is uniform and can therefore be considered an occluding disc in the view cone.

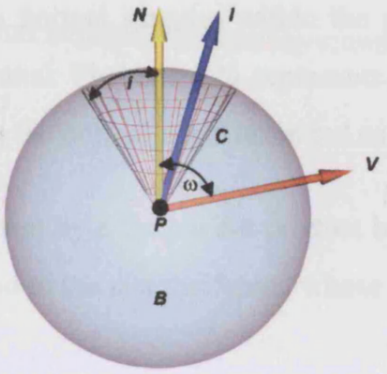
The scene node normal cone and surface area are included as hierarchical attributes in the scene nodes to support the PVA method (see Section 3.4.1 and Section 5.1). These attributes also support back face culling and hierarchical attribute weighting. The scene node's surface area is distributed over its normal cone solid angle as an approximation of the orientation of the contained surfaces. A uniform distribution is assumed and used here, but alternatives may be used.

Consider an occluder scene node B shown in Figure 50, centred at position P, with normal cone C comprised of normal vector N and angle i. The view cone axis vector V is viewpoint dependent. This will be the viewing direction assumed for the parallel projection of B's surface area.

Even though the surface area is distributed over the normal cone solid angle, the amount of surface area visible under a parallel projection is view direction dependent. Each amount of surface area associated with a normal vector I in cone C will fall off to a

cosine with increasing angle from the view vector V. Regions of the normal cone that are facing away from the view vector are discarded and considered not visible from the back.

FIGURE 50. Scene node with normal N and cone angle i, viewed from V



To find the total surface area component to the view vector, the method must take into account each normal within the normal cone, its share of the total surface area according to the assumed surface area distribution function and then how much of this surface area is component to the viewing direction. All of these results are summated to result in a value of the amount of B's surface area visible from V under a parallel projection.

This process can be described symbolically using the following definite double integral as functions of the cone angle i and view angle ω (shown in Figure 50):

$$A(i,\omega) = a_t \int_0^i \int_0^{2\pi} \Psi(E(\omega), C(i,\theta,\phi)) D(i,\theta) d\phi d\theta \qquad \text{(EQ 31)}$$

Where a_t is the total area in the scene node, $\Psi(V, I)$ is a visibility function that states the fall-off of surface area in normal vector I, viewed from V. $E(\omega)$ constructs a sample view vector with view angle ω. Function $C(i,\theta,\phi)$ generates the vector I in C at spherical polar angle (θ,ϕ). D is a radially symmetric surface area probability density function over C's solid angle with the unit constraint:

$$\int_0^i (2\pi \sin\theta) D(i,\theta) d\theta = 1 \qquad \text{(EQ 32)}$$

(EQ 31) is reformulated to a discrete form:

$$A(i,\omega)=\frac{a_l}{g}\sum_{n=1}^{g}\Psi(E(\omega),C(i,n)) \quad \text{(EQ 33)}$$

A set comprised of g random normal samples within the normal cone is created by $C(i,n)$ that returns the n'th normal. The term $1/g$ represents a uniform probability density function $D(i,\theta) = 1/g$ over the normal cone, taken out as a constant.

The visibility function Ψ is given by a simple dot product between view vector V and sample normal vector I, clipped to the near halfspace whose partition plane's normal is the view vector V:

$$\Psi(V,I) = \begin{cases} V \cdot I, \text{ if } V \cdot I \geq 0 \\ 0, \text{ otherwise} \end{cases} \quad \text{(EQ 34)}$$

Normal cone vectors are a subset of those randomly sampled in an even distribution over the unit sphere where the vector I is specified by spherical polar coordinates:

$$C(i,\theta,\phi) = \left[\sin\theta\cos\phi \;\; \cos\theta \;\; \sin\theta\sin\phi\right] \quad \text{(EQ 35)}$$

To achieve an acceptably uniform sampling distribution on the unit sphere, the polar angle θ is sampled using a probability density function proportional to the radius at θ given by:

$$r = 2\pi\sin\theta \quad \text{(EQ 36)}$$

and therefore $r \propto \sin\theta$. The sampling probability density function used is such that:

$$\frac{1}{2}\int_0^{\pi}\sin\theta d\theta = 1 \quad \text{(EQ 37)}$$

The probability density function for ϕ is uniform.

To map a uniform random value R in the interval [0...1] to that of (EQ 37), the following is solved for θ:

$$\begin{aligned} \frac{1}{2}\int_0^{\theta} \sin x \, dx &= R \\ \frac{1}{2}[-\cos x]_0^{\theta} &= R \\ \frac{1}{2}[1-\cos\theta] &= R \\ \theta &= \operatorname{acos}(1-2R) \end{aligned} \qquad \text{(EQ 38)}$$

The evaluation of the function given in (EQ 33) states how much surface area is visible from viewing direction V. For efficiency, the total area a_t can be removed from (EQ 33) and look-up tables can be used on this remaining normalized form, such that a_t can be later used as a coefficient. This finally results in the full PVA function $PVA(i, \omega)$ where i is the normal cone angle and ω is the view vector's angle to the normal cone vector:

$$\omega = \operatorname{acos}(V \cdot N) \qquad \text{(EQ 39)}$$

where ω is previously calculated for each image node because it is also required for hierarchical back face culling calculations (see angle θ in Section 5.6.2). The bi-variate PVA function's surface is shown in Figure 51 and as a plan view in Figure 52. The normalized PVA function is pre-calculated in look-up tables, the result of which then modulates the surface area in the scene node. This is then divided over the scene node's object space cross sectional area to estimate a fraction of area coverage, clipped to [0...1]:

$$OccStrength(B) = Min\left(1, \frac{a_B PVA(i_B, \omega_B)}{\pi r_B^2}\right) \qquad \text{(EQ 40)}$$

where a_B is B's hierarchical surface area and r_B is B's object space radius.

FIGURE 51. **Parallel Visible Area (PVA) Function**

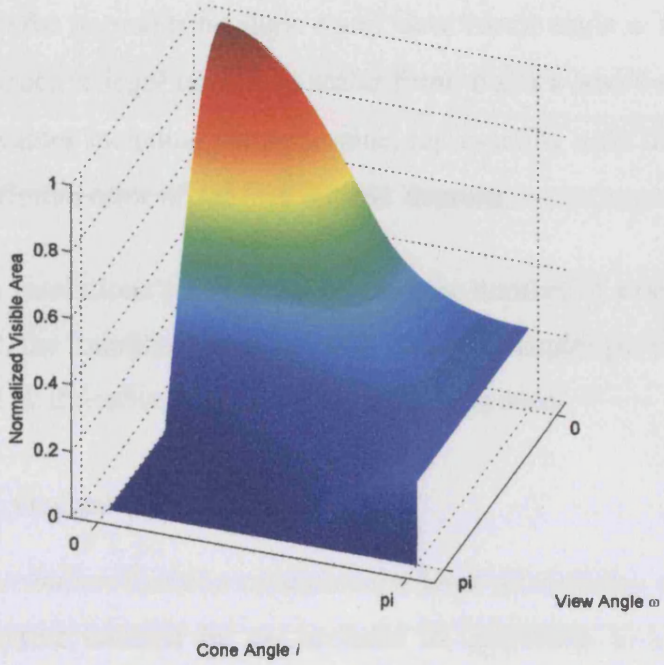
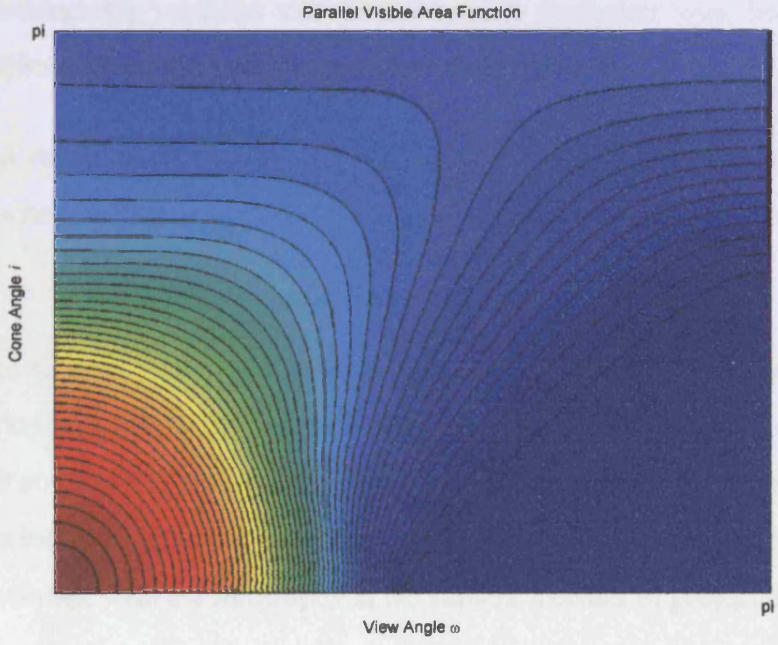


FIGURE 52. **Parallel Visible Area (PVA) Function Contours**



The PVA function $PVA(i, \omega)$ is stored as a pre-computed 2-dimensional look-up table. For simplicity, both the normal cone angle i and view vector angle ω are linearly quantized over their respective legal ranges, in scalar form, $0 \leq i \leq \pi$ and $0 \leq \omega \leq \pi$. These are quantized to 8-bit values including the zero value, representing units of $180/255 = 0.706$ degrees with a maximum error of $180/510 = 0.353$ degrees.

These quantization resolutions are specifiable, as is the number of stochastic samples to be taken. Currently, the function is sampled with 100,000 samples per cone. The look-up table is small, at 512k if double floating point precision is used.

5.7.3 Solid Angle Occlusion Ratio Function

The *occlusion contribution* function estimates the level of visibility of an occludee A that is to some extent covered by an occluder B, according to a relation with a *separate_3d* state. Relations that are intersecting in 3D do not require occlusion tests because the occluder does not make an occlusion contribution, to reduce self occlusion in surfaces. Containership relations also do not require occlusion tests, because the outcome is considered to be total occlusion of the contained node.

The occlusion contribution function in (EQ 28), reproduced in (EQ 41) results in a value that measures how well A is occluded by B as a value in the range $[0...1]$:

$$OccContrib(A, B) = OccStrength(B)OccRatio(A, B) \quad \text{(EQ 41)}$$

The occlusion strength function $OccStrength(B)$ measures the degree to which B can be an occluder, in the range $[0...1]$, considering its normal cone, surface area and direction from which it is viewed. The strength acts as a weighting for the occlusion ratio function that measures the coverage of A's view cone solid angle due to B, where the view cones of A and B overlap, with the same apex at the camera's center of projection.

The *occlusion ratio* is scalar value in the interval $[0...1]$ that states how much of the occludee's view cone solid angle is covered by the occluder, where occlusion within the occluder's view cone is considered to be uniform.

The projection of an image node's view cone on the image plane forms an elliptical conic section. Calculation of the two conic sections and their area of intersection would be both a difficult and expensive task, requiring the solution of a quartic, simply to find where they intersect.

Instead, an object space metric is used, with values resulting from a pre-computed stochastic sampling. Given two view cones, the case is mapped directly onto a look-up table occlusion value. Given an occludee view cone Q_A and occluder view cone Q_B, vectors in polar form are chosen in an even distribution over Q_A's solid angle, using the same distribution as described for the PVA function sampling in (EQ 38). If a sample vector does not lie in Q_B, A is considered visible. If the sample vector does lie in Q_B, A is occluded. The final ratio of occluded vectors to sample vectors gives the final occlusion ratio value.

FIGURE 53. Overlapping view cones of occludee Q_A and occluder Q_B with separation s

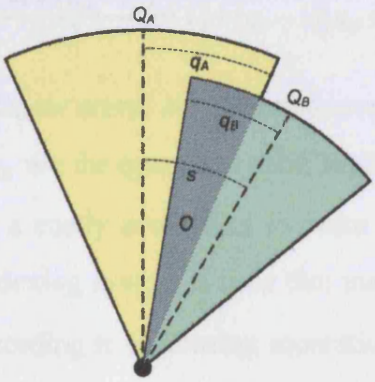


Image node view cone angles are linearly quantized to a 13-bit encoding q, giving units of $90/8191 = 0.011$ including zero and a maximum error of 0.006 degrees. This quantization is important both for reduced storage in image nodes and fast access compatibility with the solid angle occlusion ratio look-up tables that require integer values for look-up table indexing.

During rendering, many view cone comparisons will be required, to assess the occlusion ratio. The problem can be reduced to a tri-variate case of two cone angles and a separation angle between the two view vectors (view cone axes) shown in Figure 53, with overlap region O.

The occlusion ratio function $OccRatio(A, B)$ is mapped onto the more complex tri-variate function stored in a non square look-up table. The function $OccRatio(q_A, q_B, s)$ where q_A and q_B are the respective quantized cone angles and s is the quantized separation angle between the view vectors of each view cone. The same 13-bit quantization is used for s as is used for q_A and q_B for simple comparison.

A look-up table is constructed by iterating over all combinations of legal solid angles for q_A and q_B and the separation angle s, forming a cubic space. Some combinations provide a zero result, where $q_a + q_b \leq s$, indicating that the two view cones are separate.

A high resolution cubic look-up table is large in size. If zero combinations were not encoded, the size is reduced. However, this leads to a problem of mapping the quantized domain onto a linear range. It can be shown that the following performs such a mapping:

$$i = M(q_A - 1)\left[\frac{(M + (q_A - 1))}{2} + 1\right] + (q_B - 1)\left(q_A + \frac{q_B}{2}\right) + s \qquad \text{(EQ 42)}$$

where i is a result index in a linear array, M is the maximum quantized angle supported by the look-up table, q_A and q_B are the quantized solid angles and s is the quantized separation angle. This is clearly a costly evaluation to make for each comparison of two solid angles. Therefore, an indexing system is used that maps q_A and q_B to a sequential series of occlusion values according to increasing separation angle s until the first zero occlusion state is reached, which is not stored. A two dimensional indexing array is used to map q_A and q_B to the base address of the separation sequence of results. The quantized separation value s then indexes this array. In memory, the whole ratio table is a single contiguous allocation with indexes into each sequence. Occlusion ratio values are stored as 16 bit fixed point integers, giving a maximum error of approximately 2^{-17}.

To further minimize memory use, an upper bound is placed on the view cone angle that may be supported, based on the observation that occlusion ratio look-ups will only occur at certain scales in the image during rendering stage two. Currently, a limit of $M = 3$ degrees is used, placing an upper restriction on rendering level of detail threshold t_1 of 3 degrees, to guarantee that only legal comparisons will be made.

5.8 Rasterization

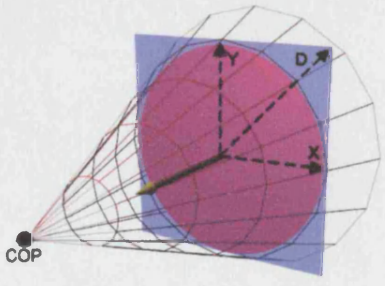
This section discusses details of the Canopy algorithm's splat based rasterization technique (see Section 2.9.5) introduced in Section 3.8. The rendering process concludes with the task of rasterizing the results of the refinement process. The result of the last refinement stage will be a large set of image nodes that are small in image space. The actual resolution that can be achieved is dependent on the hardware used. Because the current system is not greatly hardware assisted, image nodes will need to be several pixels in radius to achieve realtime frame rates. But on future hardware and in particular, with additional mappings onto modern graphics card hardware or with custom hardware, sub-pixel level detail should be achievable at interactive frame rates.

The emphasis of this research is to create a system that can provide higher resolution detail as geometry as well as approximating it using splatting. Therefore, high quality splatting has not been considered a priority.

A profile of a scene node sphere is visible, described by the image node view cone angle in Section 5.5.3 under a perspective projection. When projected onto the image plane, the footprint of the sphere is an elliptical conic section. The splat must represent this footprint in the image.

Two primary aspects are important when designing the splatting system. Over all, the system must be both fast and offer acceptable levels of image quality. One is usually traded for the other in many algorithms and splatting is another example. Mapping the splatting tasks to existing graphics card hardware helps achieve speed and quality.

FIGURE 54. View cone viewed from COP with textured splatting plane



A straight forward approach is to construct a polygon billboard, textured with the disc splat shape, whose plane normal is equal to the view vector, shown in Figure 54. In soft-

ware, the construction of four diagonal D vectors for each corner is an expensive process. An alternative mapping that is parallel to the view plane has been developed by Kilthau and Moller [119].

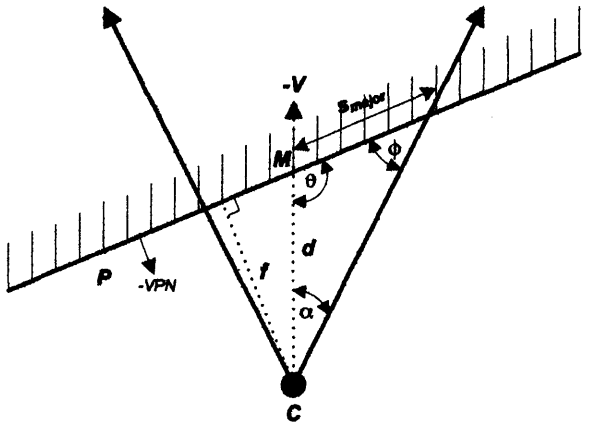
GPU based approaches can use vertex and fragment programs to evaluate splat shape more accurately, based on point or point sprite rendering (see Section 2.9.5.7). For simplicity, the current approach uses simple opaque splat using the OpenGL point primitive.

5.8.1 Conservative Conic Section Geometry

A simpler point based splatting system is used, because of the expectation of decreased reliance on multiple pixel splats as hardware performance increases. OpenGL points are used to render splats. Each circular point in image space is a conservative approximation of the elliptical conic section formed by the image node's view cone and the camera's projection plane. The size of the point in image space is calculated in software, but the projection from world coordinates, through the pipeline is performed in hardware. OpenGL points are sized and rendered by the CPU, but faster results can be achieved using a GPU vertex program calculates and sets the nVidia Cg PSIZE semantic [60] and potentially a fragment program that rasterizes the elliptical splat shape.

Given a view cone semi angle α with view vector V that is intersected by the view plane P with plane normal VPN, an ellipse is formed on P with semi-major axis length s_{major} that will be used as the circular point splat's radius, shown in Figure 55.

FIGURE 55. View cone / image plane conic section with ellipse semi-major axis s_{major}



The elliptical conic section's semi-minor axis is given by:

$$s_{minor} = d\tan\alpha$$

Angles θ and ϕ are given by:

$$\theta = \text{acos}(V \cdot (-VPN)) + \frac{\pi}{2} \quad \text{(EQ 43)}$$

$$\phi = \pi - (\alpha + \theta) \quad \text{(EQ 44)}$$

The point of intersection M between the view vector V and the plane P is the centre of both the ellipse and the point disc that will approximate it.

Given the camera model's focal distance f from the centre of projection C to P, the distance d between C and M is:

$$d = \frac{f}{(V \cdot -VPN)} \quad \text{(EQ 45)}$$

Using the edge length to opposite angle ratio equalities:

$$\frac{s_{major}}{\sin\alpha} = \frac{d}{\sin\phi}$$

the length of the semi-major axis s_{major} is given by:

$$s_{major} = \frac{d\sin\alpha}{\sin\phi} = \frac{d\sin\alpha}{\sin(\pi - (\alpha + \theta))} = \frac{d\sin\alpha}{\sin(\alpha + \theta)} \quad \text{(EQ 46)}$$

The resulting value of s_{major} is then used to define the point's radius in image space.

The most significant negative aspect of using a disc is that thickening may occur at object silhouettes, where the point discs do not represent the edges sharply, but generally points are small in image space and hardware in the near future should be capable of refining to pixel or sub-pixel levels.

5.8.2 Screen Space Refinement Thresholds

Each refinement stage (see Section 3.3.7 and Section 5.4) requires thresholds, such as t_1, t_2 and t_3 measured as view cone semi angles from the view cone axis. These thresholds specify a minimum size and the actual sizes used may be smaller, dependent on the scene tree hierarchy. It is possible to evaluate s_{major} in (EQ 46) for every image node that undergoes refinement and instead use a maximal s_{major} as a threshold. However, this is more expensive than simply having an upper bound view cone angle t used for the whole image and simply testing that $\alpha \leq t$, although this causes a small degree of conservative thresholding that increases with field of view.

To this end, it is useful to calculate t based on a maximal s_{major} image space criteria, for example, relative to pixel size. Given a view cone with angle α, its elliptical conic section is smallest when projected in the center of the image and largest when projected furthest from the center. In rectangular displays, this is a corner of the image.

Given values for a maximal view cone case of d and θ and a specified maximal image space semi major axis size g, solving (EQ 46) for α with g as s_{major} gives:

$$t = \operatorname{atan}\left[\frac{g\sin\theta}{d - g\cos\theta}\right] \qquad \text{(EQ 47)}$$

where t is the maximal threshold angle in radians to satisfy (EQ 46) where $s_{major} = g$. This approach sacrifices speed for over conservative refinement that will be most effected in the center of the image where image nodes may be slightly smaller than they need be. A relaxation may be useful, e.g. using a maximal view cone with axis passing through the top center or side center of the image, to form a conservative disc in image space, outside which, splats may be larger than the specified threshold, or just at the image center as a general best case threshold, with small error over the image.

To establish a relationship between image space threshold g and screen space pixel sizes, assuming generalized image space dimensions of $[-i_x \ldots +i_x, -i_y \ldots +i_y]$, with screen dimensions of $[p_x, p_y]$ pixels, the minimum pixel dimension in image space is given by:

$$m = min\left(\frac{2i_x}{p_x}, \frac{2i_y}{p_y}\right) \quad \text{(EQ 48)}$$

The image space threshold g can then be set as $g = rm$ where r is the radius in pixels. Achieving pixel precise results however, is not straight forward. The exact number of pixels rasterized as may vary somewhat depending on the splat primitive used, whether anti-aliasing is enabled and how the underlying graphics API and hardware choose to interpret position and size in the context of discrete sampling.

5.8.3 Basic Splatting Method

The current splatting system is simple, implemented using OpenGL points. Initially, points were unfortunately resized outside calls to glBegin() and glEnd(), inducing a small reduction in performance. This method has been temporarily used to visualize results in Chapter 6, but is resolved using the approach described in Section 5.8.4.

5.8.4 Faster GPU Cached Splatting

The algorithm may be speeded up by taking advantage of temporal coherence between frames and the limited size of the potentially visible set of image nodes identified by the occlusion culling system (see Section 3.7 and Section 5.7). A GPU based caching extension described in Bull and Slater [24] has been developed for static scenes, that constructs caches representing image nodes resulting from the expensive stage three refinement, initially with one cache associated with each image tree node in the occlusion mask image graph resulting from stage two. Image based thresholds specify a safety volume for each cache, within which, the viewpoint may move and still have all nodes in the cache projected within their tolerance size in image space. Caches are validated and reused if still valid, inherited when traversed through in stage two, or new caches are rebuilt if caches have become invalid. This reduces the bandwidth required for geometry transfer to the GPU and allows the GPU to render at higher speeds. View cones and conic section ellipses must be calculated for each image node on the GPU when rendering a cache, due to viewpoint changes. To improve frame rate consistency, a speculative cache rebuild system is also used. Results of this addition will not be covered in this thesis, but early results show an approximate 4x speedup in the whole system.

5.9 Compression

Representing complex scenes at very high resolutions requires large amounts of memory for a scene or locale. The first version of the system showed the feasibility of level of detail control, occlusion culling and other processes using the rendering techniques presented. However, memory was clearly an issue that limited substantial scene construction and testing. Therefore, two compression techniques were introduced.

The first of these is the compression of the scene node attributes listed in Section 5.1, making the description and storage of complex scenes more feasible in current hardware, though at the expense of speed due to decompression requirements.

If scene tree data structures store scene nodes in a basic way, many links are required and leaf scene nodes will store null pointers. The second compression technique reduces the number of scene tree links required. A small performance penalty is paid due to slightly increased processing.

The scene tree is embedded in a scene graph representation. These scene graphs can be read into memory to change locale or add detail and even traversed from disk on demand. This section will introduce the scene node and scene tree compression methods. This section will be described without knowledge of the scene graph embedding scheme (see Section 5.10).

5.9.1 Scene Node Compression

Each scene node (see Section 3.4.1 and Section 5.1) stores the set of attributes *position, radius, normal cone, diffuse colour* and *surface area.*

If these attributes were conventionally stored using triple vectors of single precision floats (12 bytes), and floats (4 bytes) the attribute set would total:

$$4(3_{Position} + 1_{Radius} + 3_{Normal} + 1_{ConeAngle} + 3_{Colour} + 1_{Area}) = 48 \text{ bytes}$$

This does not include additional information required in scene nodes, such as pointers to children. If alpha is also included, the diffuse colour would become 4 floats, giving a total of 52 bytes.

Using a simple rule of thumb, an object consisting of a million samples would consume 48MB of memory just to store each node's data. This amount may not appear prohibitively large by current standards of typically 4GB of memory in workstations, but many objects may require substantially more than just a million samples for pixel level detail. There may also be many such objects in a scene. A locale with just 100 objects will exceed this limit. Using the 86 bit (11 byte) compressed representation described here, this reduces to about 11MB for a million samples, a near 5 fold decrease in memory required.

Differences between memory access speeds and central processing unit (CPU) cache speeds are substantial, so it may also be of benefit to reduce bandwidth to main memory using compression.

Level of detail control processes will have a strong part to play in reducing the amount of data needed in a locale, but allowing higher resolution detail to be fetched on demand requires increased implementation complexity and increased delays for accessing data.

The next sections will detail scene node attribute compression schemes that have been developed and look at the reduction in memory that they bring.

Surface area remains the only attribute left un-compressed because substantial accuracy and wide ranging values dictate that floating point values be used.

5.9.1.1 Position and Radius Delta Compression

Each scene node stores its position and radius. The scene nodes in an originally constructed binary scene tree are positioned and sized such that a parent sphere wholly contains its two child spheres.

A delta compressed, lossy quantization method is used, equivalent to that in QSplat by Rusinkiewicz and Levoy [171] [172]. It encodes a child relative to the position and radius of its parent.

Given a parent sphere p and child sphere c, both are normalized such that the parent becomes a unit radius sphere at the origin, with the scaled child contained. The child's normalized position has offset vector V_c from the origin, defined by:

$$V_c = \frac{P_c - P_p}{r_p} \quad \text{(EQ 49)}$$

where P_c and P_p are the child and parent positions respectively in world coordinates and r_p is the parent's radius. The child sphere's radius in the normalized parent sphere is given by:

$$r_k = \frac{r_c}{r_p} \quad \text{(EQ 50)}$$

where r_k is the radius coefficient, and r_c the radius of the child.

A set of possible quantization positions exists within the unit radius sphere in a regular grid distribution such that the child is quantized to the closest legal position V_q within the sphere:

$$V_q = nearest(Q_p V_c) \quad \text{(EQ 51)}$$

where Q_p is a position quantization coefficient. The function $nearest()$ locates the closest of the eight neighbouring quantization points in the grid that is also contained by the unit sphere. This incurs the least positional error in the unit sphere, given by:

$$\varepsilon = \left| V_c - \frac{V_q}{Q_p} \right| \quad \text{(EQ 52)}$$

To guarantee that the volume of the quantized sphere is a super set of the child sphere, the radius of the child is increased by the same error distance:

$$r_\varepsilon = r_k + \varepsilon \quad \text{(EQ 53)}$$

The child's radius coefficient is then quantized:

$$r_q = roundup(Q_r r_\varepsilon) \quad \text{(EQ 54)}$$

where Q_r is a radius quantization coefficient representing the number of quantizations along the parent's radius. The radius quantization is rounded up to conservatively bound the un-quantized child sphere.

All legal quantized positions and radii of the child within the parent unit sphere are enumerated and stored in look-up tables for fast encoding and decoding. Encoding can be carried out using a simple multi-dimensional array that maps to the enumeration from V_q and r_q. Decode tables simply map back from the enumerated form to scalar position offset and radius coefficient that when multiplied by the parent's real radius, recover the offset vector and radius such that the child can be reconstructed:

$$P_c = P_p + r_p \frac{V_q}{Q_p} \quad \text{(EQ 55)}$$

$$r_c = r_p \frac{r_q}{Q_r} \quad \text{(EQ 56)}$$

In practice, the de-quantization is already present in look-up tables. Child nodes are compressed relative to their decompressed parent, rather than the real parent in the hierarchy that has not undergone compression. This stops quantization errors from propagating down the hierarchy. This is a primary reason why the compression method must cater for child spheres that are not in contact with the parent, even though the parent may have been calculated as a closest fit.

This compression system can not comply to the constraint that the decompressed child is fully contained within the decompressed parent. The only way to combine a positional quantization incurring error ε by which the radius is increased such that it is still in contact with the parent sphere, is if the child is quantized along the radial vector V_c. Unfortunately, other schemes that may attempt this such as polar coordinate based quantization schemes will still incur error that is not totally along V_c.

Even though the child containment constraint is not met in the decompressed spheres, all spheres are a volumetric super set of the original hierarchy that does conform to this containment constraint. Rendering and scene tree traversal algorithms have been developed in the knowledge that this situation can occur.

The general worst case error occurs when the offset vector V_c is equidistant between eight neighbouring quantization points, whose diagonal distance on the grid is given by:

$$\varepsilon = \frac{\sqrt{3}}{2Q_p} \quad \text{(EQ 57)}$$

The worst case radius error in the child will be greater due to the upward quantization. Therefore it has a worst case error $\varepsilon_r = \varepsilon + 1/Q_r$.

The choice of quantization coefficients Q_p and Q_r is dictated by the number of legal child states they make possible and the level of acceptable error. The number of legal states imposes a minimum number of bits required to store the largest enumeration.

Like QSplat [171], we also use quantization grid size of 13 as a reasonable trade-off between accuracy and storage size, giving a candidate set for legality testing for containment within the parent of 13^3 positions. This gives a quantization coefficient value of $Q_P = (13-1)/2 = 6$ as the number of possible quantizations symmetrically available either side of the origin and a worst case general error of $\sqrt{3}/12 = 0.1443$. An assumption that the mean error is half the worst case, gives a mean error of $\sqrt{3}/24 = 0.0722$.

The worst case radius error is $0.1443 + 1/13 = 0.2212$. The mean radius error by the same assumption is $0.0722 + 1/(2x13) = 0.1107$.

The number of legal child nodes we have found differs from a value of 7621 reported in [171] such that 8721 are identified. This requires a 14-bit encoding.

This contrasts with the $4(3+1) = 16$ bytes, or 128 bits required to store the position and radius using single precision floating point values, which is more than a 9 fold reduction.

5.9.1.2 Normal Cone Compression

Geometric compression methods developed in recent years usually include methods for normal vectors. The representation of normals using floating point triples is generally quite wasteful as such high accuracy is not required for rendering purposes [49]. For lossy compression techniques that are based on enumeration to index sets of possible

states, the problem is one of mapping quickly between a given normal vector and the enumeration. Generally, such methods quantize the vector to a nearest possible state. An important consideration is the distribution of the approximating vectors. Deering presents a high quality method [49] where octants in the unit sphere are subdivided into sextants representing normals, offering an 18 bit encoding. Deering's empirical results suggest that results are not visually distinguishable beyond separations between normals of 0.01 radians, resulting in a normal sphere consisting of approximately 100,000 normals. A similar method is presented by Taubin and colleagues based on the subdivision of the octahedron. Another method has also been presented by Slater [190] using a discretization of the Voronoi of each sample point on the unit hemisphere created through subdivision of half an octahedron.

The normal cone (see Section 5.1.2) is an un-capped positive cone with angle i measured from its central axis, oriented about a specified normal vector N. Conventional methods using single precision floating point values would require 16 bytes to store the normal vector and angle.

The method presented here is based on that of Baptista [12] and uses a 16 bit (2 byte) representation to store a normalized vector using a lossy method. Compression is carried out separately on the angle and normal vector.

The method is based on the constraint that the normalized vector has unit length, such that $x^2+y^2+z^2 = 1$. Given two of the values, say x^2 and y^2, the third value z^2 can be retrieved. Three bits are used to represent the sign of each value, leaving 13 bits to represent the normal.

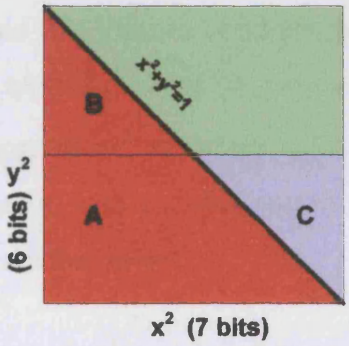
The values x^2 and y^2 are stored in a quantized form. A 2D table with x^2 against y^2, containing their respective z value is used, shown in Figure 56. All values beyond the diagonal $x^2+y^2 = 1$ are invalid as they are not normalized.

If 7 bits are used to encode x^2, this only leaves 6 bits to encode y^2 resulting in region B being un-addressable. Region A is addressable. However, some of the space addressed by x^2 is unused in the region C where $x^2+y^2 > 1$. To recover the address space lost by the

missing bit, the table's z values in B are re-mapped to C by a perceptual rotation of $180°$. All valid values with 7 bit precision are then addressable by a 7-bit and 6-bit combination. The resulting table just encodes regions A and C.

Mapping between the normal vector and the enumerated set represented by the union of ranges A and C is very fast, simply requiring quantization or re-mapping. Look-up tables are used to decode the enumeration back to a floating point triple representing the normalized vector.

FIGURE 56. Normalized vector table with non linear axes



The seven bit quantization value gives a component accuracy of $1/127$. However, the error is not distributed evenly, showing primarily where the unit sphere of normal vectors is perpendicular to an axis, showing bands due to sparse sampling.

The normal cone angle is represented using an 8-bit quantized integer, giving 256 possible cone angles to distribute over $[0...\pi]$. This contrasts with the normal cone angle used by QSplat [171] where a 2-bit representation is used, representing just four angles. The angles enumerated in their system are $\text{asin}(1/16)$, $\text{asin}(4/16)$, $\text{asin}(9/16)$, $\text{asin}((16/16))$ where the emphasis is on back face culling only where it is claimed that only a 10% over estimate is made compared to precise cone angles.

The choice of higher 8-bit accuracy is intended to serve occlusion culling metrics (see Section 5.7) that require a higher resolution description of the semi angle of the cone.

The distribution of vectors formed by this method is not as uniform as that of Deering [49] or Slater [190], but is simpler to implement, has small look-up tables and is fast to execute.

5.9.1.3 Colour Compression

The scene nodes encode RGB values, or alternatively, RGBA values to include alpha. RGB as floats would normally require 12 bytes, or stored as 3 integer bytes, 24 bits. RGBA as floats requires 16 bytes, or stored as 4 bytes, 32 bits. To reduce this, we encode RGB as a 5:6:5 (16) bit encoding. Alternatively, we encode RGBA as 5:6:5:4 (20) bits, using bit packing in the scene node description, to allow use of unused bits in words.

5.9.2 Scene Tree Compression

If a scene tree of scene nodes is stored as a conventional binary tree, the tree node stores both the data representation and two links to child tree nodes, usually implemented as CPU address width pointers, typically 32-bit on most current systems. Newer 64-bit CPUs are emerging that support 40-bit[1] or 44-bit[2] addressing in physical and virtual forms. A binary tree stored simply, with n nodes, requires $2n$ child pointers, 2 for each node, including NULL pointers at the leaves.

Parent links, from child to parent are also included to permit upward traversals, e.g. for locale management systems. These further increase the number of pointers required for each node to $3n$.

In a system where memory conservation is of importance, an implementation e.g. in a 32-bit architecture would use twelve bytes (96-bits) to implement links. Relative to the storage of compressed attributes, it would contribute a substantial amount of the over all storage requirement, with 86 bits for attributes, versus 96 bits for links.

Leaf nodes in the scene tree will have null pointers. This is particularly inefficient due to the large number of leaf nodes.

The QSplat system [171] [172] uses a hierarchy where each node has up to four children. In this sense, the system is less versatile than a binary system because an intermediate level of detail is not present. Parent pointers are not included as upward traversal is not required.

1. AMD Athlon 64, Opteron
2. Intel Itanium & Itanium 2

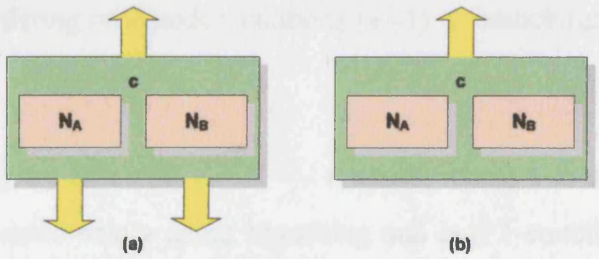
Their encoding method is also resident in memory in the same format as on disk, in a breadth first order. This enables low levels of detail to be accessed or transferred first, based on access patterns when traversing down the tree. Groups of up to four nodes are encoded in blocks, with each node indicating the number of children it has, using a 2-bit encoding. A single pointer is then shared by the group, pointing to the start of their child blocks. Pointers are not included at the leaves. The authors expect a mean branching factor of 3.5 in their assessments.

5.9.2.1 1-Container Scene Graph Nodes

To reduce the number of links in the scene tree and to remove requirements for null pointers at leaves of the tree, a system has been developed that uses container nodes to store small local regions of the tree without any internal pointers between the contained scene nodes. Links, if required, then exist solely between scene node containers. The simplest container stores two nodes, based on the constraint that a scene tree is constructed with exactly two children or are leaf nodes. This technique is scaled up to include larger containers.

To begin, consider a container node c that stores just two scene nodes N_A and N_B representing a subset of one level of nodes, shown in Figure 57(a). A parent node of some form can include a link to both child nodes N_A and N_B with just a single pointer, rather than two.

FIGURE 57. Single height 1-containers (a) branching and (b) leaf



Each node N_A and N_B must themselves, be capable of linking to two child nodes, unless they are null. If an assumption is made that these child nodes also both belong to a single container node, likewise, only a single pointer is required for each pair.

An up-link to the parent is also included, shared by N_A and N_B. In practice, null parent links will be rare, such that there may only be one at the top of a hierarchy.

In practice, null child pointers will be common and will account for half of all pointers in the hierarchy. To optimize this case, a second type of leaf container is introduced that has no child pointers, shown in Figure 57(b). The branching container type (a) can be used for internal nodes and containers (b) used at the leaves of the hierarchy.

If container types are to be distinguished, some form of type information is necessary. For example, in this C++ implementation, virtual functions are used. Therefore, all container classes have an ID or class pointer that permits object types to be distinguished at run-time. This mechanism is used to recognize specific containers. The class ID incurs additional memory overheads, typically a class pointer in C++, but is considered worthwhile because it is incorporated into the general scene graph structure where class IDs are incorporated in all scene graph nodes to distinguish between a wide range of node types at run-time.

A conventional representation of the case represented in (a) requires six pointers including parent up-links. The case shown in (a) requires just three. Normally, case (b) would also require six, but here, requires only one. Assuming a pointer length ID is used, as is the case in C++ for virtual class IDs, these efficiencies are reduced by one pointer each for (a) and (b).

This scheme using 1-containers is at best, approximately twice as efficient. A conventional scene tree consisting of n nodes, contains $(n-1)/2$ branching nodes and $(n+1)/2$ leaf nodes.

Consider a perfect binary tree with $n = 2^{h+1} - 1$ nodes, where h is the height of the tree. If this tree is represented solely using branching and leaf 1-containers, discarding the root node because it is not paired, the number of branching containers is:

$$\frac{1}{2}\left[\frac{(n-1)}{2} - 1\right]$$

with the number of leaf containers:

$$\frac{1}{2}\left\lceil\frac{(n+1)}{2}\right\rceil$$

Therefore, the number of pointers required is:

$$\frac{1}{2}\left\lceil\frac{p_b(n-1)}{2}-1+\frac{p_l(n+1)}{2}\right\rceil$$

where p_b is the number of pointers required by branching 1-containers and p_l the number required by leaf 1-containers. The efficiency ratio over a conventional tree is therefore:

$$\frac{pn}{\frac{1}{2}\left\lceil\frac{p_b(n-1)}{2}-1+\frac{p_l(n+1)}{2}\right\rceil}$$

where p is the number of pointers required by a standard tree node. In this case, with $p_{b=(3+1)}$ and $p_l = (1+1)$ to include virtual class IDs:

$$\frac{3n}{\frac{1}{2}\left\lceil\frac{4(n-1)}{2}-1+\frac{2(n+1)}{2}\right\rceil}=\frac{6n}{3n-2}\rightarrow 2$$

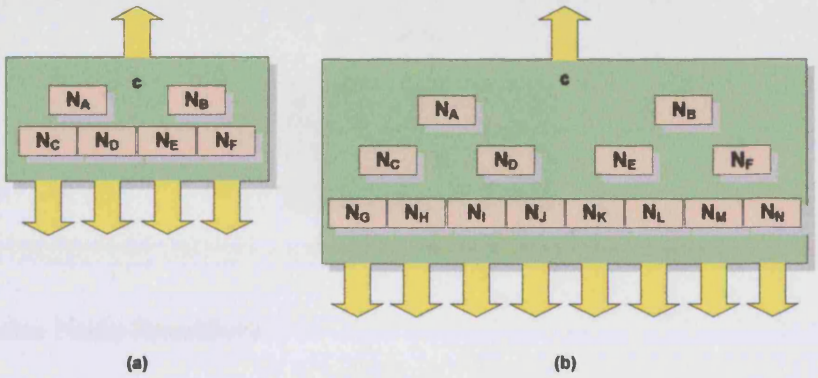
The perfect tree is the best case scenario, as some branching containers may have null pointers. This in itself can be catered for by extending the algorithm, but currently the system just includes containers for tree subsets that are perfect trees.

It is arguable that inclusion of the ID in these calculations is unwarranted because it lends itself to general versatility in the scene graph architecture. Moreover, bit-wise IDs would be more efficient. If the ID is not considered in the calculation, the ratio tends towards 3. However, a full ID is included in these calculations to treat the method in isolation of other design features.

5.9.2.2 n-Container Nodes

The 1-container is a base case for a generalised n-container system that implicitly represents all links within an extracted scene sub-tree. Figure 58 shows two further examples of perfect binary sub-trees in branching containers. Similarly for each, a leaf container is also defined without child links. A container's *root* is defined as the single scene node that refers to the container as its parent, but is not itself included in the container.

FIGURE 58. (a) Branching 2-container and (b) Branching 3-container



All links between scene nodes within the container are implied. With greater depth, comes greater levels of optimization.

The root scene node of the entire scene graph in the system is treated as a special case. An assumption is made throughout the rest of the system that a link between containers will always refer to two child nodes.

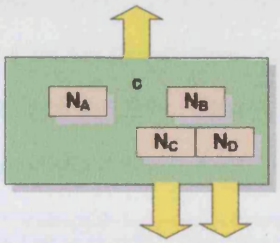
A container hierarchy of both branching and leaf versions are automatically created for each scene node type that registers with the system. Compressed and uncompressed scene node formats are used, therefore, containers are made available for both compressed and conventional scene node representations. These containers are then automatically integrated into the rest of the architecture. The use of C++ template meta programming makes such design decisions feasible and memory efficient. Currently, the system supports up to 12-containers for any specified scene node types.

The system could be extended to skew sub-trees, an example of which is shown in Figure 59 for a 2-container. Many combinations are possible at greater heights. Containers could also utilize a variable number of child pointers, with each pointer associated with a

bottom level container scene node by a bit flag. Support for these variants can be added in future as the method is very extendable.

The hierarchy of containers is integrated into the scene graph system. This representation in memory is also replicated on disk in the SCT format (see Section 5.10.4 and Section A.4) for efficient storage. The scene graphs can also exist partially in memory and be enhanced later on demand by traversal of the version on disk.

FIGURE 59. Skew sub-tree 2-container



5.9.2.3 Scene Node Specifiers

Once scene nodes are packed into containers, pointers to the contained scene nodes are of little use because given a particular node, the operations to retrieve a parent or child node are procedural. Pointer arithmetic could be used, but would rely on memory packing assumptions.

Therefore, a form of handle called a *scene node specifier* is used to refer to a particular scene node. It contains a pointer to the container and an index to the particular scene node. All further container operations can be performed using internal indices. When a child or parent is requested that is external to the container, it is returned as a specifier from the stored pointers if the container is a branching type.

The scene nodes in a container are likely to be in compressed form, though this is not necessarily the case. Therefore, the scene node specifier also acts as an interface from which the scene node compression schemes are abstracted.

Traversal of the scene tree in the scene graph is completely abstracted for higher level client functions for rendering, shadows and collision detection. Compression issues, transforms, instancing, inlining from disk, read on demand and procedural generation are all hidden (see Section 5.10.2).

An additional pointer to a traversal status object is also included in the specifier, to set the refinement in context, e.g. with respect to transforms and other push down attributes.

5.9.2.4 Container Packing

A scene tree is constructed and then compressed using the n-container system, based on the availability of a range of perfect n-containers, for heights $[1...m]$. Packing with skew containers is complex and therefore only perfect containers will be considered.

The problem is to minimize the total number of containers used, subject to the constraint that every node must be belong to exactly one container. Containers become more memory efficient with each increase in tree height. A game tree describing the search space can be traversed, as shown in the pseudo code in Figure 60.

FIGURE 60. Perfect n-container exhaustive search space traversal

```
traverse(n)
{
        for C = each legal container with root n
        {
                for a = each bottom scene node of C
                {
                        traverse(a)
                }
        }
}
```

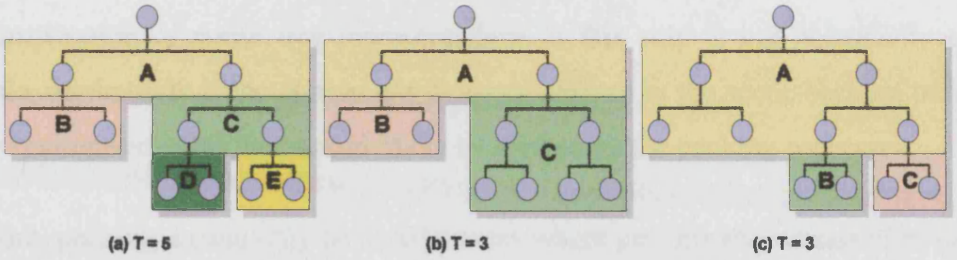
A small example of all possible packings for a scene tree is shown in Figure 61. The total number of containers T is shown for each. Case (a) is the most inefficient, using the most containers. Cases (b) and (c) are equivalent and therefore, more than one optimal solution can exist. The search space quickly explodes. For trees of height h, the perfect tree form has the largest number of possible packings. For example, a perfect tree with a height of just 7 has approximately 10^{15} packings. The number of packings for a perfect tree is the series given by the recursive definition:

$$f(H) = \sum_{L=1}^{H} f(h-L)^{2^L} \qquad \text{(EQ 58)}$$

with the stopping condition:

$$f(1) = 1$$

FIGURE 61. Possible perfect n-container packings of the same scene tree



(a) T = 5 (b) T = 3 (c) T = 3

5.9.2.5 Greedy Packing Algorithm

A greedy packing algorithm is used, packing from the root downwards. Given a scene tree with root node N, it is traversed downward starting with the first child pair until one or more nodes in level L has no children. An $L-container$ is then created for this perfect sub-tree.

The same algorithm is then executed for each scene node at level L forming a recursive system. The example in Figure 61(c) is an example packing that would result from this greedy method, where in this case, it reached an optimal solution.

Scene tree construction from a source set of nodes to a container tree hierarchy can be performed in two stages. In stage one, the scene nodes are partitioned forming a hierarchy of scene nodes. The results of this process is stored without any pointers because each tree level can be processed sequentially, such that a parent in one level can easily be associated with child nodes in the next. This greatly reduces the amount of additional memory required during the construction phase. Simultaneously, the greedy packing system constructs the hierarchy of containers. Once the downward recursion has completed, the unravelling recursion back up the tree calculates the hierarchical attributes of the new internal branching nodes of the scene tree.

In stage two, a second pass down the tree levels packs each scene node into the correct container ready for use (see Section 5.3).

5.9.2.6 Re-packing

The compression of scene tree representations in this way is not necessarily always desirable, particularly if the system has dynamic regions in the scene because trees may require realtime rebuilds that would likely be slowed by the packing processes.

Therefore, packing should only be used in cases where performance issues of re-packing due to tree changes is not a problem, or prior knowledge is available that scene regions will be static.

Regions that are known to be highly dynamic may be represented using only 1-containers such that no packing is required.

It is conceivable that heavily packed tree regions could be unpacked for some period of time to cater for fast changes in regions of a scene and later re-packed when dynamics have ceased. This is also the case for scene node compression, where for reasons of speed, un-compressed scene nodes could be used for some period of time before reverting back to a compressed form.

Re-packing algorithms are left as future work.

5.10 Scene Graph Representation

A geometrical description of the scene is provided by the scene tree, described in Section 3.4. The basic binary scene tree on its own is not efficient with respect to memory use, or as versatile as conventional rendering APIs such as VRML97 [203], Java3D [193], Open Inventor [209] and SGI Performer [188] in terms of control.

Therefore, to allow the system to be technically competitive with other conventional systems, methods were sought to address memory and versatility issues that may hinder common, practical use.

Scene graphs in rendering APIs have been popular since the early nineties, themselves extending from early, simple hierarchical structures for object representation. Apart from geometric description, they offer ways of controlling aspects such as materials and dynamics. In particular, instancing and referral to external data in backing store are also generally provided. This type of functionality would suit the scene tree system well, as it reduces memory substantially for multiple copies of objects.

Scene graph architectures are generally extensible node systems that usually have a strong relationship with a textual language in a file format like parse trees, though this is not essential.

To provide more versatile scene definition and control, compression and procedural scene definition framework, research has been undertaken into embedding the scene tree into a scene graph architecture. A *working set* of the locale scene graph can be maintained in core memory, with extra detail fetched into memory on demand in a manor transparent to client applications that traverse the scene graph or scene trees. It is also possible to traverse and render directly from backing store. The scene graph may be spread out among many files, which are also accessed transparently.

For rendering and other client algorithms, scene tree traversal is transparent of the scene graph architecture (see Section 5.10.2).

The n-container data structures described in Section 5.9.2 are a subset of a wider range of scene graph nodes. Object IDs or class pointers can be used to distinguish between nodes, with defined sets of polymorphic functions for generalized traversal operations.

This section will give an overview of the general scene graph framework. It will describe how the system is centered around the concept of procedural generation, how this representation can be maintained in backing store and its fetching on demand.

The scene graph architecture is designed such that it is superficially similar to other scene graph systems, but greater complexity exists at lower levels to provide this abstraction, such as scene tree construction for group nodes. A particular departure is the concept of *generator* nodes that provide a hierarchical procedural generation framework.

5.10.1 Generators - Hierarchical Procedural Generation

Some module in the system must be responsible for generating scene graphs, particularly from designed or scanned models. It would appear useful if this system was open and extensible, to allow various ways for scene graphs to be created. Also, rather than have the modules responsible for creating scene graphs external to the data structures they generate, it would be beneficial if they were embedded within the data structures. This enables their states to be represented in close association with their scene graphs.

A range of scene graph nodes are responsible for generating a sub-graph. These will be termed *generators*. Large, highly detailed scenes will typically have many objects that are perceptually constructed from a number of other objects. Although this is a conventional property of 3D scene design, it suggests that hierarchies of generator nodes would be suitable. It may also allow scene detail to be taken to much higher resolutions than those obtained by sampling models, to obtain real 3D geometric textures, rather than those of finite texture maps that we currently rely on.

A generator may be any procedural system that creates a sub-graph, ranging from large scale scene composition, to microscopic surfaces. Generators can convert scenes, read previously converted scenes or employ any other procedural system of choice. Level of detail control will ensure that rendering processes will not request more detail than is required for image generation.

The creation of the scene is considered a run-time process that executes on demand, such that there is no formal pre-processing. Whilst this may incur unwanted delays at some times, it provides an open system with reproducible creation and historical design accountability within one data structure.

The range of available generators could conceivably be extended to include software plug-in modules offered in a completely open way which can be simply achieved using operating system executable object models such as Microsoft DLLs, COM or the .Net model, but this is left for future inclusion in the architecture which currently relies on generators developed solely within the system.

Turing computable, interpreted languages for performing specific tasks could also be encapsulated within generators, where specific scripts are maintained as custom data.

Generators may choose to generate a scene graph once and cache its scene graph for reuse, or generate each time on demand. To enable this functionality, generators are notified when they are traversed through by any client algorithm and low level traversal systems abstracted from rendering or other client algorithms, automatically request further detail from the generator if child nodes are not present.

It is conceivable that through suitable locale management with solutions to finite word length issues associated with position and scale, hierarchical procedural generation may offer scenes that are truly infinite in both size and detail.

Generators must be responsible not just for generating their scene graph, but also for handing over the production of further detail to generators at the leaves of their graphs. These will be termed *sub-generators*. In a philosophical sense, they should always be capable of doing so, but this is difficult to achieve in a practical way and therefore, null generators must be tolerated, generally implied by leaf container nodes or null children in branching containers.

A generator node is referred to by a parent scene node. Due to the design of the container link optimization system described in Section 5.9.2, all scene graph links typically refer to two child nodes. Therefore, a parent scene node will expect to refer to two child nodes

in the generator, rather than a root node. Therefore, the root node of the generator is considered to be a *virtual root*.

Because a particular generator may have a specific scene node requirement, the creation and management of nodes is left to the generator. It follows that if a generator is responsible for the creation of its scene graph, it should also be at least partially responsible for its destruction when required.

5.10.2 Scene Graph Abstract Traversal

The scene trees that represent the geometry and attributes of a scene are embedded in a scene graph structure. Scene graph nodes control structure and properties of the scene, providing specific, additional information for use when a region of a scene is traversed by algorithms for rendering or analysis tasks such as collision detection.

A trade-off exists when considering the formulation of higher level client algorithms that use the scene graph to carry out their task. Scene graph nodes provide versatility in scene description and dynamics, but offer a far greater range of entities that must be dealt with by any client algorithm. This is a problem that may deeply affect their complexity and ease of development.

To resolve this issue, an abstraction is provided by a *scene node specifier* handle (see Section 5.9.2.3). The specifier is a form of smart pointer that refers to a scene node in a container. However, using specifiers, the architecture is capable of abstracting the more complex underlying affects of scene graph nodes to make it appear that the algorithm is traversing a simple binary scene tree. Examples of aspects that are hidden include:

1. Scene node decompression
2. Scene tree decompression (containers)
3. Instancing and Inlines
4. Arbitrary affects of scene graph nodes
5. Attribute transforms, e.g. during instancing
6. Requests for generator detail when scene tree terminals are reached
7. Fetching of more scene graph detail in generator caches in backing store on demand.

These functions are performed in co-operation with a traversal state management system that stores a stack of traversal states, with the most recent state shared by all nodes when

traversing downward until affected by a scene graph node that wishes to affect the state, at which point a new state is created and pushed onto the stack for the child graph. States are popped from the stack when conceptually traversing back up the scene graph.

Given a scene node specifier to a scene node, the specifier can provide the node in the normal, un-compressed form. Requests can simply be made for the parent or children of the node, resulting in one or two scene node specifiers respectively. All details of the underlying scene graph architecture are invisible, offering much simpler operations for client algorithms.

Scene graphs can also be traversed normally, accessing each scene graph node hierarchically.

5.10.3 Conversion Generator

Conversion generators import scene data in an external format and construct a scene graph with embedded scene tree that describes the scene. Container based link compression is optional, where compression is not desirable if the scene is likely to have dynamic components.

The conversion generator used in this thesis imports and translates polygonal geometric scenes. A VRML97 rendering engine has been developed, allowing a specified scene graph in one or more VRML97 files to undergo the voxelization (see Section 3.4.3 and Section 5.2.2) and scene tree construction (see Section 3.4.4 and Section 5.3).

A generator file cache is created that caches the generator's state with its constructed scene graph that is reused on subsequent traversals.

5.10.4 Generator File Caches

Generators that use expensive methods to create their scene trees, such as conversion generators are likely to want to cache the results of their scene graphs.

Facilities have therefore been developed, for generator scene graphs to write their graphs to a binary format named SCT (Scene Tree). Generators can also then be in charge of reading the SCT file caches on demand, controlling the number of scene tree levels read

in each cache access. When developed, any generator can inherit these abilities if required.

If the cache is not found, the full generation process is invoked and the results once again written to a file cache.

See Section A.4 for further details on the SCT format and out of core rendering.

5.10.5 Group Node

Group nodes are present in most scene graph APIs and are vital for constructing objects hierarchically as compound objects. A group node has one or more child graphs. In this case, the group node will have one or more generator nodes as children. An example is shown in Figure 62 for four child generators.

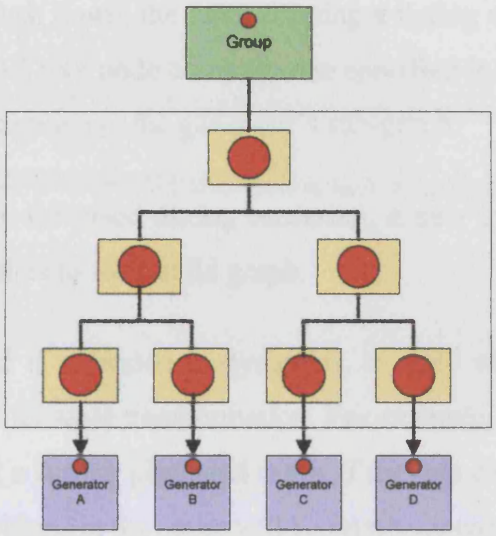
Creating a group node for this form of hierarchical system is substantially more difficult than those of conventional APIs. Each child member of the group is a generator node with its own scene graph, with embedded scene tree. If the group represents the union of these child scene trees, to obtain a hierarchically consistent scene tree, the group node must generate its own scene tree that instances the child generators. The group node is therefore a type of generator node.

An example group scene tree is shown in the box in Figure 62. Group nodes will serve as aggregate levels of detail of the child objects that they represent.

If a group's child scene trees may be dynamic, the scene nodes used can be an uncompressed form, in 1-containers which require no packing.

Each generator has a concept of a *virtual root* node, shown in Figure 62 as a small node. This is a node that may or may not be stored, that represents the root node of its scene tree. This is necessary because a single root node can be required, e.g. to communicate root node properties up the scene tree, but due to the scene tree compression methods described in Section 5.9.2, a link to a generator is implicitly linking to two of its child nodes because each link in the system refers to two children.

FIGURE 62. Group node scene tree with child generators



All child generators must have hierarchically consistent attributes before the group can be calculated, so that all positions, sizes and attributes are calculated based on the correct states of its children.

It may be possible for group node updates to be *lazy* or *strict*. In a lazy update, changes in the child generators are merely reflected by absorption of changes in attributes up the scene graph. This is made possible by allowing scene nodes to intersect or contain each other without unique spatial occupancy. The structure of the scene tree does not change. In a strict update, the structure and attributes of the scene tree are fully recalculated upward to the locale scene node.

5.10.6 Instance Node

To create large scenes, particularly when using very complex models, it is often desirable to instance scene graphs in specific contexts to reuse them to reduce memory costs. Instance nodes in conventional scene graph APIs simply refer to a scene graph. The only additional requirement of this system is that specific details be given as to the size and position of the instance, such that the child graph can be transformed to fit inside an *instance root* node specified within the instance node.

Various options can be made available for instancing. These include filters for whether node size is *pull up* or *push down*. Pull up attributes are passed up the scene graph unfil-

tered, where as push down are filtered to affect the instanced graph. Colour can also be specified as pull up or push down, the latter creating a tinting effect by re-targeting the generator's existing virtual root node colour to one specified in the instance node, using colour space transforms applied to the generator's sub-graph.

When instance nodes are traversed during rendering, a new traversal status is created with a transform that applies to their child graph.

It could also be optional if extended to dynamics, to state whether an instance node dynamically recalculates its scale transformation. For example, consider the case where two cars are instanced at a certain place and scale. If the two cars drive away from each other, their scale and positions in the scene will be relatively correct. However, if scale is dynamically recalculated to fit the child graph into the instance root node, the cars will shrink as they separate. This effect may be desirable in other situations, for example when scaling objects which instance 3D texture detail.

5.10.7 Inline Node

Inline nodes allow scene graphs to be referred to and read from backing store. They have been designed in the system to be a form of instance node, with the additional behaviour that they fetch their child graph from a specified SCT file. The root scene graph node in every SCT file is a generator that creates the child graph. This generator is read, created and executed. If a cache of the generator's scene tree exists, it is also included in the file.

5.11 Hierarchical Consistency and Scene Dynamics

This section discusses hierarchical consistency issues with respect to the scene tree's scene node attributes. In particular, it addresses cases of scene tree construction, scene tree instancing, inlining and scene dynamics.

Support for dynamic scenes is not yet developed in our implementation, but this section discusses related issues.

It is important that the locale's scene tree be *consistent* with respect to its spatial hierarchy and hierarchical attributes. This means that all scene nodes in the scene tree are correct and up to date.

5.11.1 Hierarchical Attribute Consistency

Constraints are imposed on the system by the scene tree's requirement for a parent to contain its child nodes, which dictates the parent's position and radius. Moreover, scene node attributes such as the normal cone, colour and surface area must be hierarchically correct. Hierarchical consistency must be ensured within the locale when it is first set up for use and then if surfaces or objects move.

Primarily due to the surface area and normal cone angle attributes, it is not trivial to instance an arbitrary sub-generator that entirely conforms to the invoking parent scene node's attributes. Transforms can be applied to position, radius, normal cone vector and colour, but surface area and normal cone angle are harder to conform to. Therefore, the working set in the locale is made hierarchically consistent, upwards, from the bottom to the top.

A generator's scene graph can only be considered *consistent* if all the hierarchical attributes in its sub-graph are correct up to its virtual root node (see Section 5.10.5). A generator's sub-generators must first be consistent before the generator's own consistency can be calculated. Attributes of the virtual root in the sub-generator are requested and propagated up the scene graph to the generator's own virtual root.

5.11.2 Attribute Pull Up and Push Down

Attributes can be propagated in one of two ways, either *attribute pull up* or *attribute push down*. Pull up attributes are propagated up the scene graph unfiltered, forcing parents to approximate the attributes, where as push down attributes are filtered to affect the sub-graph.

For example, an instanced child may have its own position, radius and rotation. This may be integrated 'as is' into the invoking scene graph using the pull up method to keep its existing dimensions, or positioned, scaled and rotated to fit into a scene node specified when instanced using the attribute push down method. Normal cone angle and surface area must typically remain pull up attributes, so the invoking scene graph is still affected.

Attributes that can be pull up or push down include position, radius, normal cone vector and diffuse colour. Generally, the normal cone angle can not use push down, because this would require some form of non-intuitive warping of the instanced generator's scene tree. Push down surface area is also generally not used because this conflicts with scale. Procedural generators may be developed that create mathematically defined environments that are capable of conforming to normal cone angle and surface area push down specifications, but this will not be considered here.

Colour can be specified as pull up or push down. Pull up simply integrates colour into the surface area weighted averaging system up the scene tree, but push down uses a tinting effect that re-targets the instanced scene graph's virtual root node colour to one specified in a way that affects the whole sub-tree to change its colour scheme. In this case, we use a simple RGB colour space transform that conforms to the surface area weighting in the child nodes. Components that map out of the unit colour space are clipped.

Because generators can be instanced multiple times, all push down effects must be calculated at runtime during traversal, so that no actual attributes are modified. Therefore, whenever a generator is traversed through, a new view dependent traversal state is associated with the generator and its sub-tree that contains contextual information to be applied to the sub-tree. This context includes push down transforms for position, radius, colour and inverse square surface area transform. This is invoked by scene node specifiers rather than image nodes, to permit traversal by a wider range of client algorithms.

5.11.3 Dynamic Consistency

Dynamics are not demonstrated in this thesis, but this Section discusses related issues of hierarchical consistency for dynamic objects or surfaces.

A scene tree results from a spatial partitioning process. The use of group nodes is a convenient way to add scene components, without complete spatial partitioning of the entire scene. It is a compromise that is conforming to the over all structure.

Based on the level of tolerable inefficiency in the over all hierarchy, consistency solutions may range from *lazy* updates that simply propagate attributes up the scene graph to the locale root to reflect change, to *strict* updates that completely re-partition the entire scene. Error metrics may be developed to force rebuilds when lazy update error becomes too large. All updates must be made in a way that minimizes noticeable changes. It may also be possible to specify dynamic regions of the scene, where parent generators are over-specified in their radius and normal cone, permitting dynamics within, that still conform to the parent's attributes such that change is absorbed and need not be propagated up the hierarchy.

Simple dynamics are already possible in the system, where group nodes with moving child objects are dynamically spatially re-partitioned. However, a more comprehensive approach is required to realize a more automated, flexible and robust architecture.

Further complexity arises because changes may occur outside the locale working set, at higher levels of detail or elsewhere that affect the locale. Such issues require substantial analysis to derive an effective solution and are out of the scope of this thesis.

5.12 Summary

This chapter has given further details on how to implement the algorithm introduced in Chapter 3. Firstly, it has covered the scene node data structure, how they are sampled from scenes and constructed into a scene tree, with container packing and geometrical compression.

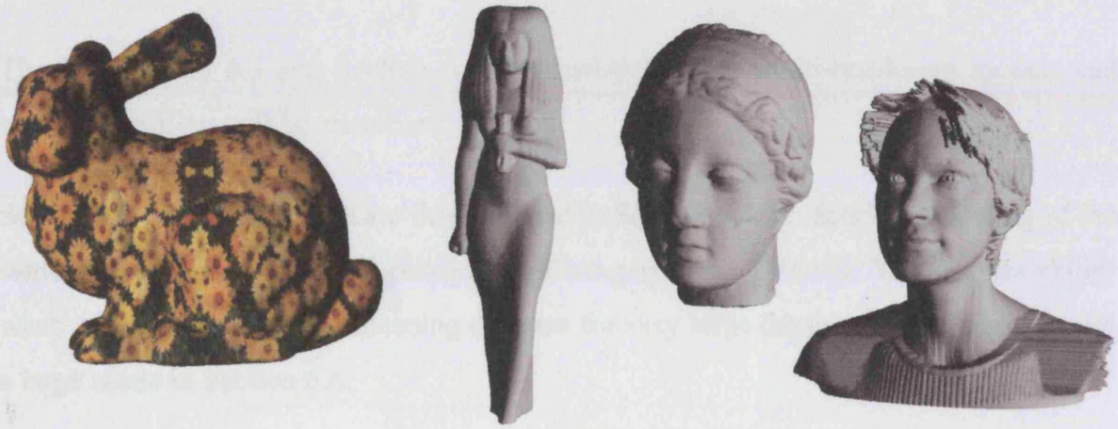
The main rendering function was then given, with pseudo code for each of the main stages, which implement different functionality at different image scales. Further to this, specifics of image graph refinement were examined for front to back refinement ordering and occlusion culling, with pseudo code on how image relations are established and classified.

Hierarchical view volume and back face culling were then detailed, which restrict the rendering traversal to the view volume and to only the front visible surfaces of polyhedra. Occlusion culling was then covered, detailing the occlusion mask, the PVA function for occluder strength estimation and solid angle occlusion ratio function that combine to make occlusion contributions. These contributions to occluders are then summated to inform the main occlusion culling decision based on a thresholding scheme. Rasterization using splatting was then covered, using a disc that conservatively covers the elliptical conic section formed by the view cone of a scene node and the camera's image plane.

Scene trees are hierarchically compressed using a quantized delta encoding for position and radius and quantized normal and normal cone angle. Scene tree compression uses containers that represent local, full regions of the scene in an internally pointerless representation. The container hierarchy is constructed using a greedy packing system simultaneously with the scene tree's construction. The generalized scene graph representation allows the scene tree to be embedded in a more powerful data structure, common to most graphics APIs including *group, inline* and *instance* nodes in a basic procedural scene tree generation architecture. Out of core rendering was touched on and is discussed further in Appendix A.4.

The need for hierarchical consistency and updates for dynamics are then discussed, defining *pull up* and *push down* attributes and lazy and strict updates to maintain consistency.

Chapter 6 Results



This chapter tests aspects of the algorithm for correctness, scalability and performance.

An ***overview*** of tests is given in Section 6.1, commencing with ***scene sampling*** and ***translation*** details in Section 6.2 for a range of 3D models. ***Level of detail*** renderings are tested in Section 6.3. ***Occlusion culling*** correctness is then examined in Section 6.4.

Scalability with increasing ***depth complexity*** is compared against a standard polygon rendering engine in Section 6.5. A ***large scene walkthrough*** is then compared against standard polygon rendering in Section 6.6. A ***massive scene*** is then rendered in Section 6.7 to obtain performance figures. Occlusion culling speed-up is tested for a ***dense scene*** in Section 6.8, comparing against rendering without occlusion culling. Approximate ***depth ordering*** examples are then compared against standard z-buffering in Section 6.9.

Additional features of ***collision detection*** and ***hierarchical shadow mapping*** are tested in Section 6.10 and Section 6.11.

Results are approximately ***compared against other systems*** in Section 6.12 and the chapter's results are overviewed in a ***summary*** in Section 6.13.

6.1 Overview

Having described the algorithm and implementation details in the last three chapters, this chapter will now discuss the results. The first sections will examine whether the algorithm works as intended, with visualizations of its level of detail and occlusion culling functionality. Later sections will then look at how well the algorithm works in terms of performance, finishing with other functionality tests for depth ordering, collision detection and shadows.

In Section 6.2, scene sampling will be discussed, with several examples of medium to high resolution scanned 3D models. Their properties will be looked at, with an analysis of their pointer compression efficiency and computation times.

Then, in Section 6.3 and Section 6.4, the qualities of the multi-resolution models and occlusion culling will be examined.

Scalability and performance are then covered in Section 6.5, comparing rendering of the same scenes with a standard polygon rendering system. This test looks at scalability when adding objects with increasing distance for very large depth complexities and with a large scene in Section 6.6.

Massive scale scene rendering is then examined in Section 6.7, with scenes that would conventionally contain over 200 Billion polygons, being shown to render at near interactive frame rates.

Occlusion culling speed-up is assessed for a densely occluded scene in Section 6.8, comparing a fast, simple point based refinement algorithm with no occlusion culling, against the Canopy algorithm's image graph based refinement with occlusion culling.

Depth ordering results are given in Section 6.9, comparing examples rendered using the occlusion mask depth ordering without a z-buffer and rendered normally with standard z-buffer resolved images.

Collision detection and shadows will then be looked at in Section 6.10 and Section 6.11 respectively, giving examples with performance details. Finally, results are compared with other systems in Section 6.12, with a chapter summary given in Section 6.13.

All performance data in this chapter are based on the use of one Intel 2.8GHz Pentium 4 system with 1GB PC800 RAM, 533MHz FSB, ATA-100 7200 spin speed hard drive and an nVidia Quadro4 900XGL graphics card. The operating system used is Microsoft Windows XP Professional SP2. The version of the Canopy algorithm tested uses minimal hardware support to provide clearer findings of the performance of just the algorithm itself. To this end, the version of the Canopy system tested is not pipelined (see Section 3.10) such that the graphics card is not asynchronous with the refinement stages and executes separately in the fourth stage. The *read on demand* rendering system reads all scene files into core memory and therefore no out of core rendering is carried out. All scenes are rendered using the same global locale.

All timings in this chapter are taken using a performance timer accurate on the system hardware to a resolution of $2.29x10^{-7}$ seconds, or 229 nanoseconds. Rendering stage thresholds t_1, t_2, t_3 are given in both degrees ° and pixel radius p measured with the view cone at the center of the image (see Section 5.8.2).

6.2 Scene Sampling and Translation

This section translates several high resolution polygonal models to the compressed, partitioned scene graph representation. All models are single polyhedra, but the technique is just as easily applied to scenes.

Details of the polygon model, sampling parameters and sampling results are given. An analysis of the efficiency of the resulting compressed scene graph representation will also be examined, with respect to pointer optimization. Each of the following sections will look at a particular model in detail. A summary of all models will then follow.

Not all models have colour information. This is purely because they are provided as uncoloured or textured meshes. To demonstrate colour, the Bunny model has had a texture map applied. Most models however, are left un-textured for clarity. Each model sampling is just an example of typical use. Models could be sampled at higher or lower resolutions as required. Higher resolutions are always better because the data is available, even though it may not be used during rendering. A trade-off has been made here based on memory resources required for sampling. Attempts have been made not to use virtual memory, to give more accurate figures for translation performance.

Each model has been sampled using the voxelization method, with sampling density automatically chosen based on an analysis of polygon edge lengths, described in Section 3.4.3 and Section 5.2.2. The method calculates the mean edge length μ and establishes a base sampling length as $l = \mu - k\sigma$, i.e. k standard deviations from the mean. The number of samples to distribute over this base length is then specified as γ. No clipping of l to β occurs when calculating a sampling density in these examples (see Section 5.2.2). Basic translation details are given in the following sections and additional performance and container usage data is given in Appendix A.5. Translation performance details are given for the voxelization stage and stages 1 and 2 of the hierarchical scene graph construction using container link compression with the embedded scene tree with geometry compression (see Section 3.4.4, Section 5.3 and Section 5.9). In translation stage one, the scene nodes are partitioned forming a hierarchy of scene nodes. Simultaneously, the greedy packing system constructs the hierarchy of containers. Once the downward recursion has completed, the unravelling recursion back up the tree calculates

the hierarchical attributes of the new internal branching nodes of the scene tree. During this process, scene nodes considered to be duplicates are filtered out by merging attributes to a single scene node. Therefore, the tables contain both the number of scene nodes resulting from the voxelization and the number that resulted from filtering, which is also the number of leaf nodes in the scene tree constructed. In translation stage two, a second pass down the tree levels packs each scene node into the correct container ready for use, with hierarchical geometry compression.

Times and file sizes are also included for writing to the SCT binary file format that caches the sampled model's scene graph for future use by the generator. Lastly, details are given in Appendix A.5 of the distribution of container objects used for both branching and leaf variants. Containers with tree heights of 1 to 12 are used, containing 2 to 4096 nodes respectively.

All performance figures have shown a variation of about 2% for processing tasks and about 5% for disk based tasks.

6.2.1 Bunny (Textured) [Stanford Computer Graphics Laboratory]

FIGURE 63. Bunny model with texture rendered using Canopy

The Stanford Bunny is a common test model used in the computer graphics field. It is the result of ten multiple combined range scans of a physical clay model. A simple flower texture map has been applied to demonstrate handling of colour during sampling and hierarchical construction. Basic translation details are shown in Table 8, with efficiency and performance details given in Appendix A.5.1, Table 20 and Table 21. The number of containers used of each height and type is shown in Appendix A.5.1, Table 22. This model took 18.4 seconds to voxelize and construct, with a total time of 49.1 seconds including reading the source VRML97 file and writing the SCT file.

TABLE 8. Bunny - translation details

Feature	Value
Density	$k = 3, \gamma = 2$
Original Vertices	34,834
Original Triangles	69,451
Voxelized Scene Nodes	2,221,573
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	1,401,494
Scene Nodes per Polygon (Mean)	20.18
Scene Tree Nodes (Total)	2,802,986
SCT File Size	37.23 MB
Number of Containers	636,275
Scene Graph Child Pointers (32 bit)	1,033,124

6.2.2 Igea Venus [Cyberware Inc.]

FIGURE 64. Venus model rendered using Canopy

The Venus model is an artifact held by the University of Thessaloniki, scanned by Cyberware Inc. No colouring information has been applied to keep the model plain. Basic translation details are given in Table 9, with efficiency and performance details given in Appendix A.5.2 in Table 23 and Table 24, with container height data in Table 25. This model took 23.6 seconds to voxelize and construct, with a total time of 56.1 seconds including reading the source VRML97 file and writing the SCT file.

TABLE 9. Venus - translation details

Feature	Value
Density	$k = 2, \gamma = 2$
Original Vertices	134,345
Original Triangles	268,686
Voxelized Scene Nodes	2,679,194
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	1,339,598
Scene Nodes per Polygon (Mean)	4.99
Scene Tree Nodes (Total)	2,679,194
SCT File Size	35.59 MB
Number of Containers	613,101
Scene Graph Child Pointers (32 bit)	984,676

6.2.3 Isis [Cyberware Inc.]

FIGURE 65. Isis model rendered using Canopy

The Isis model is an egyptian sculpture scanned by Cyberware Inc. of the goddess. Colour has not been included for clarity. Basic translation details are shown in Table 10, with efficiency and performance details given in Appendix A.5.3 in Table 26 and Table 27 with container height usage in Table 28. This model took 41.8 seconds to voxelize and translate, with a total time of 94.4 seconds including reading the source VRML97 file and writing the SCT file.

TABLE 10. Isis - translation details

Feature	Value
Density	$k = 2,\ \gamma = 2$
Original Vertices	187,644
Original Triangles	375,284
Voxelized Scene Nodes	5,344,016
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	2,614,887
Scene Nodes per Polygon (Mean)	6.97
Scene Tree Nodes (Total)	5,229,772
SCT File Size	67.95 MB
Number of Containers	1,083,271
Scene Graph Child Pointers (32 bit)	1,581,856

6.2.4 Female [Cyberware Inc.]

FIGURE 66. Female model rendered using Canopy

Female model scanned by Cyberware Inc. Basic translation details are shown in Table 11 with efficiency and performance data given in Appendix A.5.4, Table 29 and Table 30 with container usage details in Table 31. This model took 60.8 seconds to voxelize and translate, with a total time of 143.6 seconds including reading the source VRML97 file and writing the SCT file.

TABLE 11. Female - translation details

Feature	Value
Density	$k = 2, \gamma = 1.5$
Original Vertices	302,948
Original Triangles	605,086
Voxelized Scene Nodes	7,850,883
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	3,720,501
Scene Nodes per Polygon (Mean)	6.15
Scene Tree Nodes (Total)	7,441,000
SCT File Size	100 MB
Number of Containers	1,708,822
Scene Graph Child Pointers (32 bit)	2,766,962

6.2.5 Summary of Sampling and Translation Results

Once source models have been read, voxelization has shown to be a fast process, the time obviously dependent on the resolution and number of polygons in the source model. All examples have an average practical fill rate of approximately 1 million voxels per second, including full attribute interpolation of texture colour, vertex colour and surface normal. Vertex colour may not contribute if only textures are used. This is slow in comparison to hardware graphics card fill rates often currently quoted in the billions, but such comparisons are hard to make for several reasons. Firstly, such figures are peak for single polygon writes to video memory. The sampling process is implemented in software with no parallelism and is running on top of an operating system that also shares resources. Thirdly, various other processes are taken into account with the voxelization figure, such as mesh access, sample creation and memory management. It is likely however, that future work may concentrate on hardware support for the voxelization process. However, because the time is relatively small, the voxelization times are considered reasonable.

Duplicate sample filtering has reduced the sample set by about $1/2$ for the Venus, Isis and Female models. These samplings have a lower mean samples per polygon of about 5 to 7 that will result in a higher percentage of samples shared along edges. The bunny model, sampled with a mean of about 20 samples per polygon shows a reduced filtering ratio of about $1/3$. This function will be asymptotic, approaching zero as the ratio of samples interior to the polygon to those along the edges becomes larger.

Disregarding voxelization and I/O operations, the *Bunny*, *Venus*, *Isis* and *Female* models required approximately 16s, 20s, 36s and 53s respectively, to construct their hierarchies using container scene graph nodes. stage one of the hierarchical partitioning and container optimization process has bee observed to require the most significant processing times. This is to be expected as most of the partitioning, duplicate sample filtering, container creation and hierarchical attribute calculation is carried out in this stage. As a simple linear rule of thumb measure, the sample to time ratio for stage one varies from about 165,000 nodes per second for the Venus model, to 183,000 nodes per second for the Isis model. stage two, which encodes and packs the resulting nodes into their scene graph containers is relatively fast.

6.3 Level of Detail

This section summarizes rendering details for two models at multiple levels of detail.

6.3.1 Model Resolutions

Each model is rendered with stage thresholds $t_1 = 2.5° (73p)$, $t_2 = 1.0° (29p)$, where p is the angle's pixel radius and $t_{OccCull} = 1.9$ with t_3 set to one of a range of view cone angles [1.0°|0.5°|0.25°|0.1°|0.05°] to a 1280x1024 window. With the camera settings used, these thresholds have the following maximum splat ellipse semi major axes in pixels of [29.2, 14.7, 7.4, 2.9, 1.5] respectively, measured with the view cone at the center of the image.

Most of these thresholds would not be used in practice, but are given here to visualize the operation of the hierarchical level of detail system.

6.3.2 Results

Images are shown for each LOD in Figure 67. The splat radius means and standard deviations are shown in Table 12 and Table 13.

Note that the t_3 threshold is a minimum criteria and that the nodes actually splatted are likely to be smaller as they are the first conforming nodes in the hierarchy. Therefore, the mean and standard deviations of the actual rendered splat radii are also given, measured in pixels. Splats are also rendered in a higher density than the radius of the splats because they overlap.

The total number of rasterized image nodes is recorded, along with the frame period required to calculate the image.

TABLE 12. Bunny model level of detail

View Cone Angle t_3 (°)	Max Splat Radius (Pixels)	Rendered Splat Radius Mean (Pixels)	Rendered Splat Radius σ	Frame Period (s)	Image Nodes Rasterized
1.0	29.2	17.4	2.8	0.098	3,646
0.5	14.7	12.4	1.5	0.109	7,021
0.25	7.4	6.1	0.8	0.166	26,093
0.1	2.9	2.5	0.4	0.474	127,294
0.05	1.5	1.0	0.1	1.579	495,298

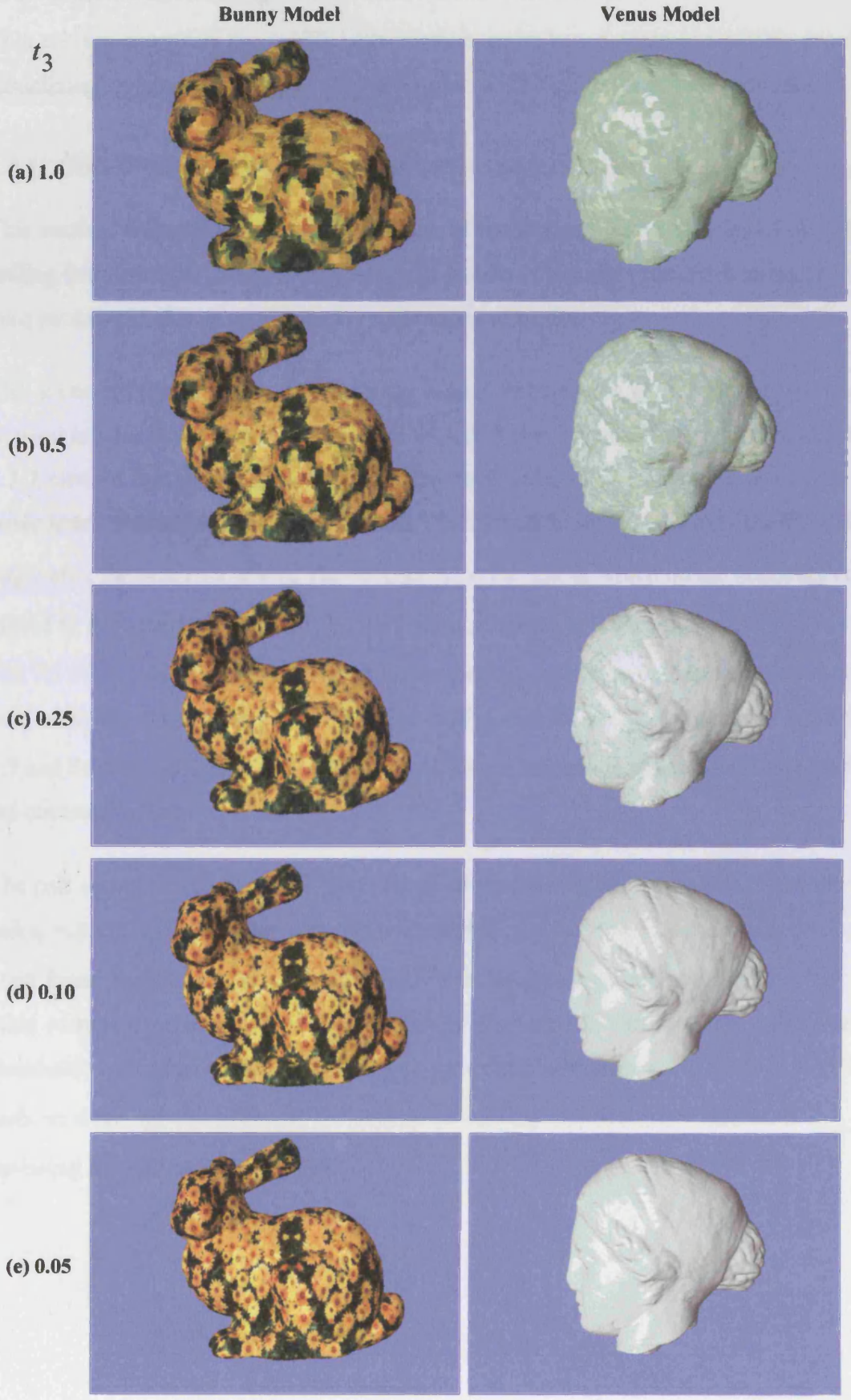
TABLE 13. Venus model level of detail

View Cone Angle t_3 (°)	Max Splat Radius (Pixels)	Splat Radius Mean (Pixels)	Splat Radius σ	Frame Period (s)	Image Nodes Rasterized
1.0	29.2	17.7	2.9	0.066	2,984
0.5	14.7	12.5	1.5	0.075	5,614
0.25	7.4	6.1	0.8	0.121	21,641
0.1	2.9	2.5	0.4	0.354	109,131
0.05	1.5	1.1	0.2	1.222	421,078

The current splatting implementation for stages 3 and 4 is quite slow, so interactive frame rates currently only start with a view cone angle of about $t_3 = 0.1° (2.9p)$, with a frame rate of about 2 fps for this level of image coverage.

Although render times are comparatively slow compared to many systems, this scene has only a single depth complexity and substantial image coverage. Rendering several hundred thousand models (see Section 6.7) results in little additional overhead.

FIGURE 67. Multiple levels of detail of Bunny and Venus models



6.4 Occlusion Culling

This section examines the results of the image graph based occlusion culling process, visualizing its operations and varying behaviour with changing threshold parameters.

6.4.1 Object Pair with Occlusion Area

This section will consider the case of a pair of models and give examples of occlusion culling between them, firstly to visualize the results of the algorithm with respect to varying parameters and secondly to give indications of performance.

The scene is constructed with the Bunny model in front, occluding the Venus model. Several images have been rendered, each with differing parameters. Recall from Section 3.3.7 onward that the three rendering stages have three view cone angle thresholds for basic level of detail control, t_1, t_2 and t_3. The first of these, t_1, is simply the view cone angle after the setup stage one. The second, t_2, is the size at which image nodes are committed to the occlusion mask. In the third stage, angle t_3 specifies the refinement resolution for further enhancement of image nodes that have not been occlusion culled in stage two, ready for rasterization. The occlusion culling threshold $t_{OccCull}$ discussed in Section 3.7 and Section 5.7.1 is responsible for specifying when image nodes are culled based on the coverage metric.

The pair shown in Figure 68 to Figure 71 are all rendered with the same stage one threshold $t_1 = 2.5°\ (73p)$ and occlusion mask resolution $t_2 = 1.0°\ (29p)$, that have empirically been found to offer a reasonable trade-off between accuracy, performance and look-up table storage overheads and will therefore not be varied in this test. The rasterization threshold t_3 in stage three is also set to a view cone half angle of $t_3 = 0.1°\ (2.9p)$. The tests in these figures examine the effects of varying the occlusion threshold $t_{OccCull}$, assessing both the images and timing.

6.4.2 Results

In the first medium resolution, medium occlusion culling threshold example in Figure 68 (a), a threshold value of $t_{OccCull} = 1.9$ is used, which has been identified to provide reasonable accuracy. Both models show no signs of occlusion artifacts due to self occlusion and no artifacts are visible on the silhouette between the Bunny and Venus behind it.

Image (b) in the figure shows a side view that visualizes the set of image nodes that were used to render image (a). Holes in the rear model show the occlusion culled regions.

Image (a) in Figure 69 shows a visualization of the occlusion mask used in Figure 68(a). Each image node is coloured based on the occlusion strength of the occluder in the mask, scaled between zero and the largest value present in the occlusion mask, which lies in the interval `[0...1]`. Red values denote high values, fading through to green at the low values. Occlusion strength is lower at silhouettes due to the fall-off provided by the PVA function.

Finally, Figure 69(b) shows image graph relations of all types that are present in the final occlusion mask, rendered as edges. White relations are intersecting_3d, separate_3d are red and contained_3d are green, though none are visible. A certain amount of squared structuring is visible due to the spatial partitioning system.

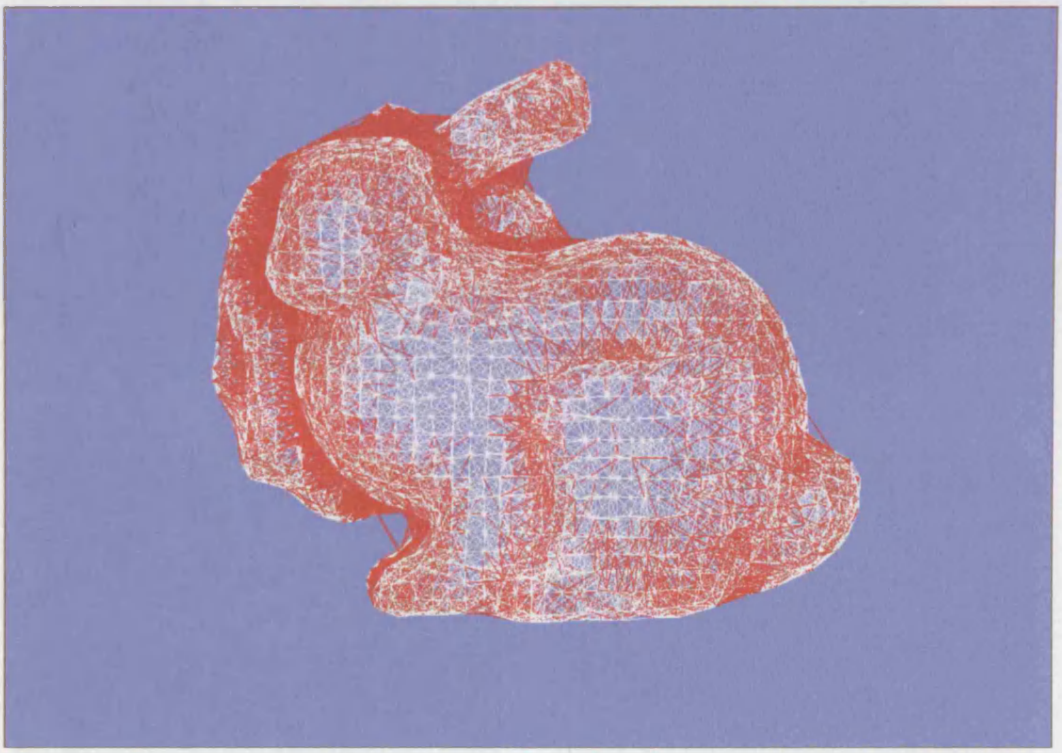
FIGURE 68. **Med res, Med Occ.** $t_1 = 2.5$, $t_2 = 1.0$, $t_3 = 0.1$, $t_{OccCull} = 1.9$

(a)

(b)

FIGURE 69. Med res, Med Occ. occlusion mask and final image graph

(a)



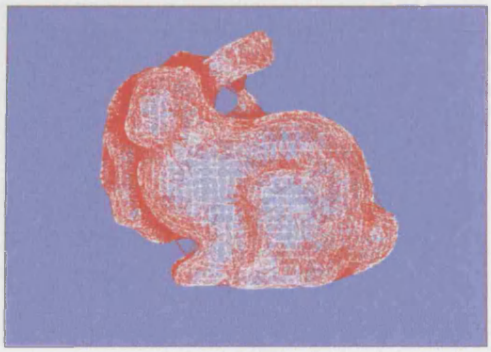
(b)

FIGURE 70. Med res, Low Occ. $t_1 = 2.5$, $t_2 = 1.0$, $t_3 = 0.1$, $t_{OccCull} = 1.0$

(a)

(b)

(c)



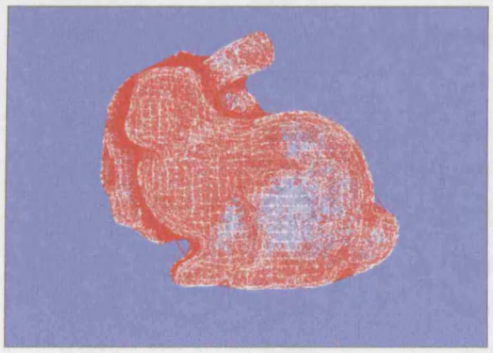
(d)

FIGURE 71. **Med res, High Occ.** $t_1 = 2.5$, $t_2 = 1.0$, $t_3 = 0.1$, $t_{OccCull} = 10.0$

(a)

(b)

(c)



(d)

The second example in Figure 70 shows a case where $t_{OccCull} = 1.0$ is low enough to begin to introduce occlusion culling artifacts that are clearly visible in image (a) as missing regions in the rear model. Again, image (b) shows a side view of the culled areas. Subtle differences with Figure 69 can be seen in the occlusion mask in (c) and image graph (d), particularly with a hole near the ears.

The third example in Figure 71 shows a case where $t_{OccCull} = 10.0$ is large enough such that no occlusion culling occurs. Image (a) can be seen to be equivalent to that in Figure 68(a), but clearly, no culling has occurred in (b). The image mask and graph in (c) and (d) will be slightly fuller due to the presence of extra nodes in the rear model that are not culled.

Rendering performance for each of these three examples is shown in Table 14. This includes timing information for the whole frame (as measured) and for each of the four stages of the rendering algorithm. The number of image nodes resulting from each stage is then given, followed by a summary of all relations formed by all stages during rendering. Relation states are only shown for 3D cases, their respective 2D states are not shown.

TABLE 14. Rendering performance for occlusion culling threshold $t_{OccCull}$ variation

Property	$t_{OccCull} = 1.0$	$t_{OccCull} = 1.9$	$t_{OccCull} = 10.0$
Time period frame (Total)	0.487 s	0.548 s	0.689 s
Time period stage 1 (Setup)	0.021 s	0.021 s	0.022 s
Time period stage 2 (Occlusion Culling)	0.067 s	0.110 s	0.167 s
Time period stage 3 (Refinement)	0.102 s	0.120 s	0.154 s
Time period stage 4 (Rasterization)	0.293 s	0.295 s	0.345 s
Nodes after stage 1	516	516	516
Nodes after stage 2	1,652	1,996	2,646
Nodes after stage 3 (Rasterized)	116,244	133,550	169,990
Relations formed in stages 1 & 2 (Total)	161,175	223,529	307,984
Relations separate_3d in stages 1 & 2	94,013	145,374	212,160
Relations intersecting_3d in stages 1 & 2	63,157	74,060	91,631
Relations contained_3d in stages 1 & 2	4,005	4,095	4,193

The frame rate for each is less than two frames per second, primarily because of the large image coverage. The majority of the frame period is used in stages 3 and 4 as may be expected, due to the large number of nodes to be processed. Reducing the image quality to lower resolutions by reducing t_3 speeds up stages 3 and 4 substantially, but a medium rasterization resolution has been chosen for these three examples.

Comparison can be drawn between the examples in Figure 68 and Figure 71, with medium and high occlusion values respectively. The first is a solution that appears to offer reasonable quality, whilst accurately culling occluded surfaces. The second performs no occlusion culling at all.

A direct comparison of frame times is not an entirely fair measure of the occlusion culling's effectiveness, because the case with no culling involves image graph calculations in stage three. In light of this argument, the speedup is 20%. Although this value appears low, considering the ratio of image nodes occluded by the bunny, to those in the unculled image, an estimate of 20% would appear feasible. It can be seen that the number of image nodes rasterized in stage four, resulting from stage three, is reduced by 36,440 representing a 21% reduction in the number of image nodes rasterized without culling. The speedup in stage three and four combined, is about 17%.

TABLE 15. High resolution, Medium Occ. threshold

Property	Value
Time period frame (Total)	1.623 s
Time period stage 1 (Setup)	0.021 s
Time period stage 2 (Occlusion Culling)	0.110 s
Time period stage 3 (Refinement)	0.452 s
Time period stage 4 (Rasterization)	1.039 s
Nodes after stage 1	516
Nodes after stage 2	1,996
Nodes after stage 3 (Rasterized)	505,639
Relations formed in stages 1 & 2 (Total)	223,529
Relations separate_3d in stages 1 & 2	145,374
Relations intersecting_3d in stages 1 & 2	74,060
Relations contained_3d in stages 1 & 2	4,095

As a fourth example, to demonstrate performance changes with level of detail threshold t_3, the same scene is rendered with $t_3 = 0.05°\ (1.5p)$ in Figure 72, with performance information given in Table 15. The primary differences between Table 15 and the medium resolution equivalent in Table 14 are the increased stage three and four times with increased number of nodes resulting from stage three that are rasterized in stage four. Image quality increases notably. Stage one and two's times remain constant.

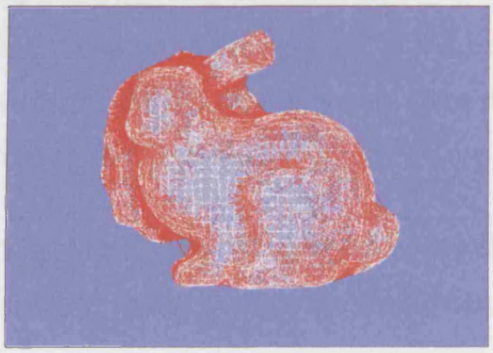
An example of occlusion culling with high depth complexity is given in Section 6.5, with other occlusion culling examples with large scenes in Section 6.6 to Section 6.8.

FIGURE 72. **High res, Med Occ.** $t_1 = 2.5$, $t_2 = 1.0$, $t_3 = 0.05$, $t_{OccCull} = 1.9$

(a)

(b)

(c)



(d)

6.5 Scalability with Increasing Depth Complexity

This section examines scalability of the algorithm and that of conventional polygon rendering with respect to increasing scene depth complexity.

The same scene is first constructed in both the Canopy system and a conventional VRML97 rendering engine, also developed within the system. Their rendering performances are then evaluated. The scene's depth complexity is progressively increased for both algorithms.

The scalability of each algorithm is then assessed and the two are then compared. This test makes use of occlusion culling with fusion, but particularly, level of detail control. All objects are in the frustum, so no use is made of hierarchical view volume culling.

Two further tests then clarify scalability of the algorithm with this scene.

6.5.1 Depth Complexity Test 1

To ensure that the amount of detail and depth complexity to be rendered is known, a scene has been constructed in both rendering systems that is entirely in front of the virtual camera, within its frustum. Initially, two instances of the Venus model are placed in front of the camera. The model on the left is behind the model on the right, with a small degree of overlap between the two in the rendered image. This is to additionally demonstrate the system's capability for occluder fusion.

To increase the scene's complexity, additional pairs of models are added with increasing distance from the viewpoint that are occluded. Pairs are added in steps of one hundred (200 models), up to 1,000 pairs, totalling 2,000 models to be rendered in the image, finally totalling a scene conventionally consisting of about 573M polygons and a depth complexity maximum of 2,000 in the overlap and 1,000 elsewhere.

Each increasingly complex scene is rendered by both the algorithm and a conventional polygon rendering system to a 1280x1024 window. Frame periods are then evaluated. The first frame in both systems is discarded. This is to allow the system to stabilize, particularly with respect to L2 cache behaviour, which has shown to result in as much as a

30% increase in the polygon renderer. This also allows the image tree to be constructed, the nodes of which are then reused in subsequent frames. As the system can benefit from frame coherence, it was deemed appropriate to allow this to occur given that this is typical usage. As will be seen, the algorithm's times are low compared to those of the polygon renderer. Therefore, these timings were subject to a mean and variance calculation over 30 frames to help discard any potential noise from operating system when attempting to compare neighbouring results with small differences. Figure 74 shows images rendered by the algorithm (a-c) and by the standard polygon renderer (d). The test view is shown in (a) and (d). The side view in (b) shows 2,000 models from the side. The regions left after occlusion culling in image (a) with 2,000 models is shown in (c). Image (c) demonstrates how the two occluders have acted together to cull all models behind the first pair. Note also that the further of the pair has also been partially culled.

6.5.2 Results

The frame period results can be compared in Figure 73, against the number of models in the view frustum. Clearly, there is a dramatic difference in scalability over these magnitudes of objects. While the polygon rendering gives a good example of anticipated linear scalability, the scene node rendering approaches about 1.2s over the first 200 objects. A magnified view of the same scene node curve is shown in Figure 75. The mean of the variances for the scene node observations is small at $3.19x10^{-5}$. Clearly, the results are strongly sub-linear.

FIGURE 73. Canopy vs. polygons - frame periods for increasing scene complexity

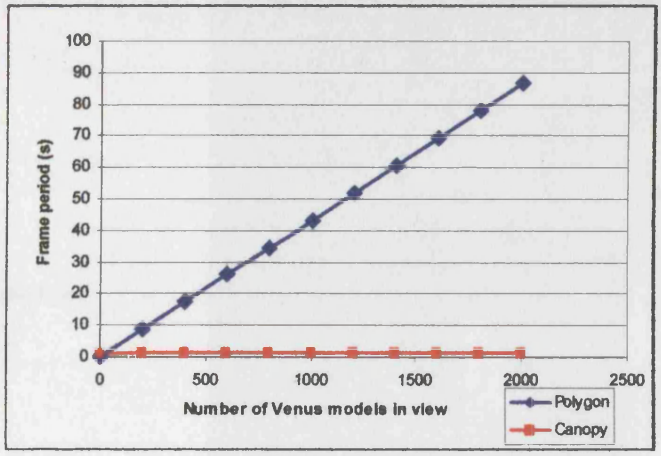
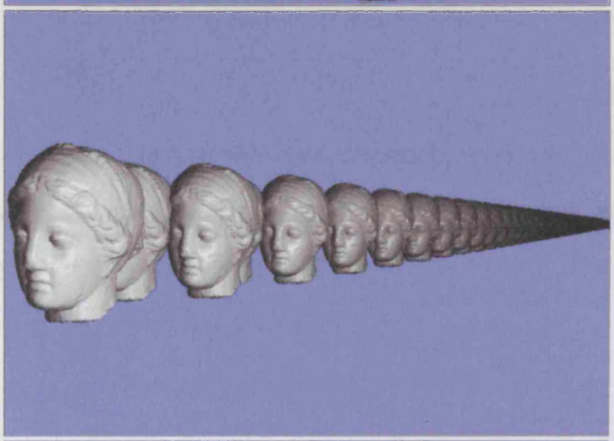


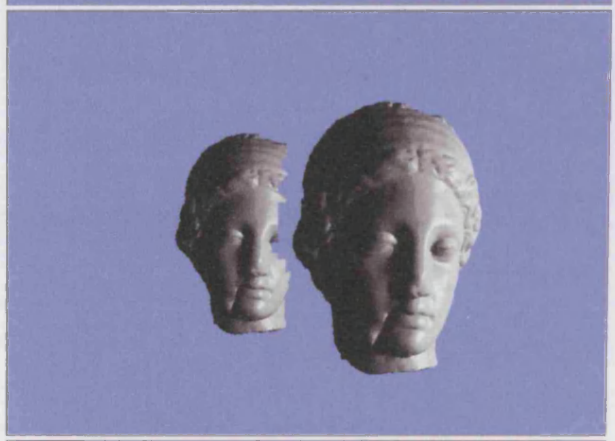
FIGURE 74. Test views (a) Canopy & (d) polygon rendering. Also with (b) side view, (c) culled

(a) Rendered image

(b) 1,000 pairs, 2,000 models



(c) Side view of occlusion culling



(d) Polygon rendering

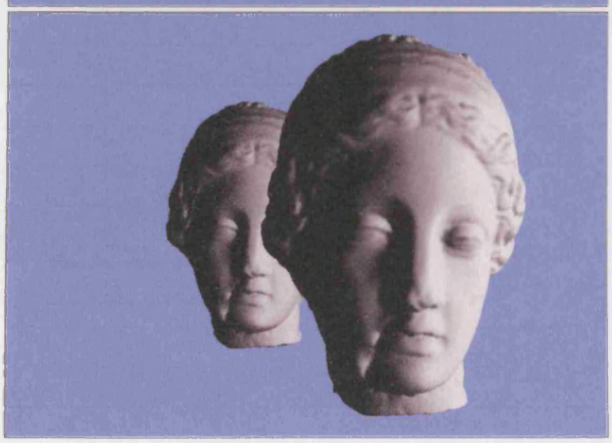
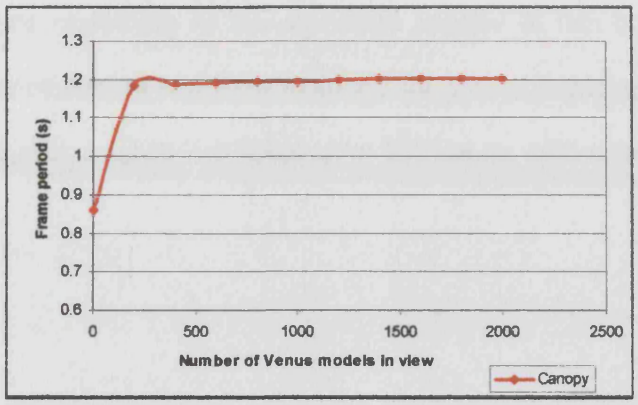


FIGURE 75. Scene node rendering, magnified



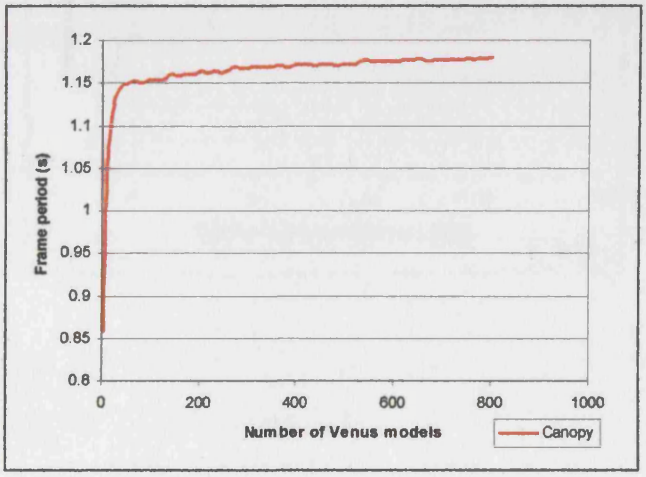
6.5.3 Depth Complexity Test 2

To give a higher resolution view of the algorithm's performance over the first 800 objects, a second round of timings was taken, again averaged over 30 frames, starting with the second frame. The polygon based rendering system was not used. The scene starts with 2 models, then 10, moving up to 800 in increments of 10.

6.5.4 Results

The results are shown in Figure 76. The mean of the variances for the observations is small at $6.13x10^{-6}$.

FIGURE 76. Scene node rendering, frame period per 10 additional models



6.5.5 Depth Complexity Test 3

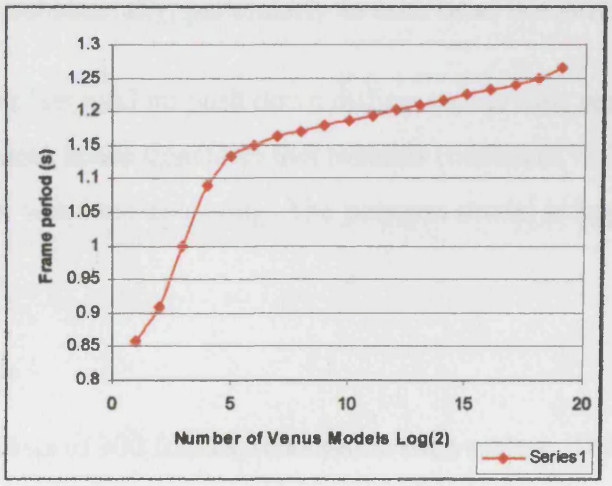
To further investigate scalability of the algorithm applied to this type of scene across magnitudes, a series of frames was then rendered for scenes containing 2^x Venus models, or 2^{x-1} pairs. Again, a mean was taken over 30 frames, with a measured mean variance of $5.86x10^{-5}$.

6.5.6 Results

The results can be seen in Figure 77, where the curve increases until about 2^5 models, entering an apparently linear phase in the Log_2 scale. The polygon rendering system was not used in this test due to time required to render.

The number of models rendered ranges from 2 to $2^{19} = 524,288$. If rendered using standard polygon rendering, these scenes would range between about 500k triangles up to the most complex scene with 140 Billion.

FIGURE 77. Scene node rendering, frame periods per additional 2^x models



6.6 Large Scene Performance Against Polygon Rendering

This section presents a case study of a short fly-through of a large scene, comparing the performance of the Canopy rendering algorithm with conventional polygon rendering without level of detail control or occlusion culling.

6.6.1 Scene Construction

A randomly jittered grid of 30x30 instances of the *female* model is used, each with a bounding radius of 0.5 and a spacing of 1.5, which is then jittered by up to 0.5 in the ground plane components.

The total number of models has been kept low at 900 to reduce the amount of time required for polygon rendering. Represented conventionally, this scene has approximately 544M polygons when instances are expanded. Represented as scene nodes, the expanded scene has approximately 7 Billion nodes.

The polygon based rendering engine scene uses spherical bounding volumes for view frustum culling. These volumes are not culled hierarchically, but this is not likely to impact performance substantially, particularly as each head is a single mesh.

The scene node scene has random push down diffuse colour tints applied to demonstrate this basic form of colour space transform that remains consistent with respect to the hierarchical surface area weighted colouring. The polygon model is kept with original uniform colouring.

6.6.2 Walkthrough

The fly-through consists of 300 frames, rendered in each system. The scene node rendering is performed using parameters $t_1 = 2.5°\,(73p)$, $t_2 = 1.0°\,(29p)$, $t_3 = 0.1°\,(2.9p)$, $t_{OccCull} = 1.9$. These have been chosen to give acceptable trade off between image quality and performance. Slightly higher quality rendering should be visible in the polygon based images.

Viewpoint-object collision detection is also enabled and performance data is given, although no collision is actually triggered during the sequence. Viewpoint collision detection thresholds are $T_r = 0.02$ and $T_d = 0.1$.

The camera starts on one edge of the grid of objects, shown in Figure 78, moving forward at object height until frame 50, where the camera is raised above object height, shown in Figure 79. The camera continues until about frame 100, where it again rises and then again towards then end at frame 150. These rises demonstrate performance changes due to increased distance. Sample images are shown in Figure 80 for frames 1, 25, 50, 100 and 200. Results of both scene node and polygon rendering are shown.

FIGURE 78. Camera fly-through, plan view of 300 frames starting near origin

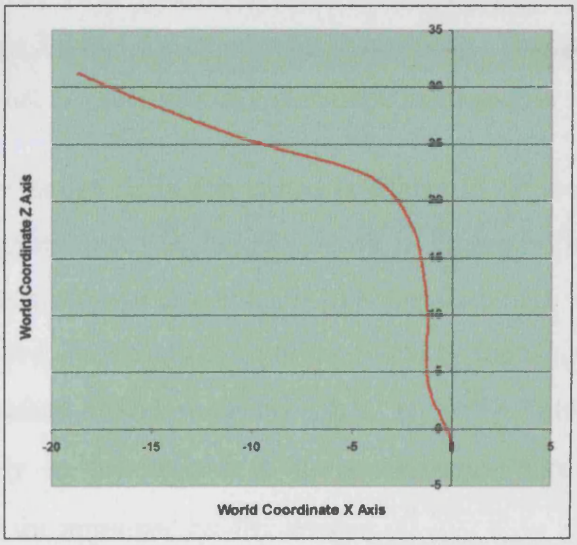
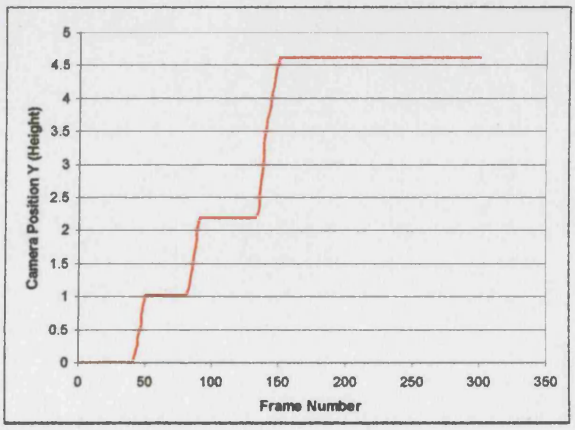


FIGURE 79. Camera fly-through height for 300 frames, starting at 0



6.6.3 Results

Frame periods are shown for 300 frames of the sequence in Figure 82(a) using a Log_{10} scale to allow both algorithms to be compared in the same graph. The frame times for both algorithms show similar performance curves using this scale. Performance generally increases throughout the sequence due to reduced numbers of large objects in the images.

The polygon rendering method starts at about 36s per frame, compared to the scene node rendering that starts at about 0.5s per frame. The mean speedup ratio of the scene node algorithm over the polygon rendering in this example is 70 times faster.

The scene node frame periods and periods of all four stages are shown in the linear scale in Figure 82(b). Stages 3 and 4 are expectedly most costly. The number of nodes resulting from stage three that are subsequently rasterized in stage four is shown in Figure 83.

The number of image nodes occlusion culled is shown in Figure 84(a) with the mean view cone angle of culled nodes in degrees shown in Figure 84(b) for each frame. The number of occluded nodes drops dramatically at about frame 50, where occlusion levels are dramatically reduced, such that no nodes are culled by the algorithm. The mean view cone angle of occlusion culled nodes is high, typically between $0.6°\,(17.5p)$ and $1.3°\,(38p)$, particularly in comparison to the rasterization threshold $t_3 = 0.1°\,(2.9p)$, which could be lower for improved quality images.

FIGURE 80. Walkthrough frames 1, 25, 50, 100, 200 for scene node & polygon rendering

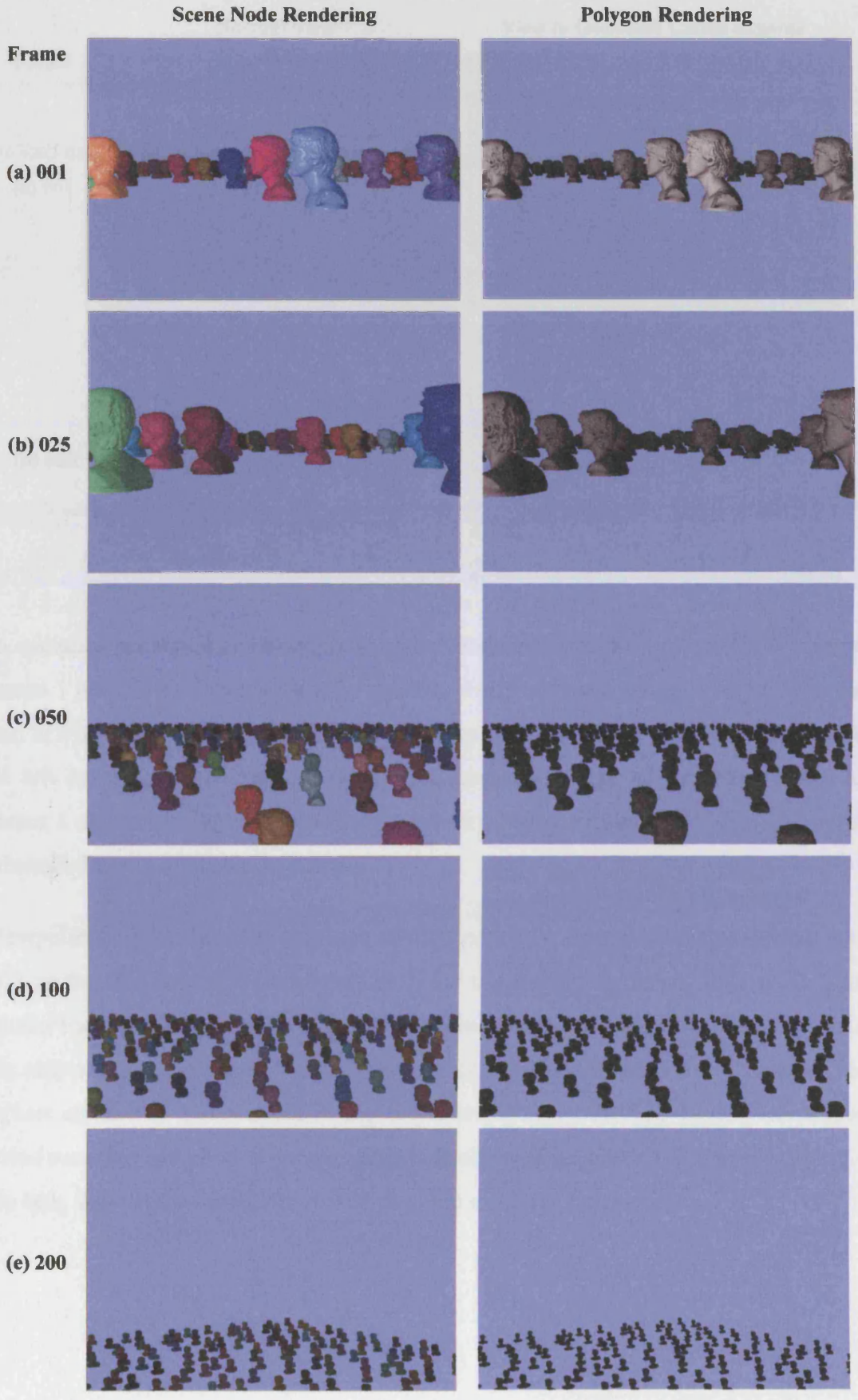
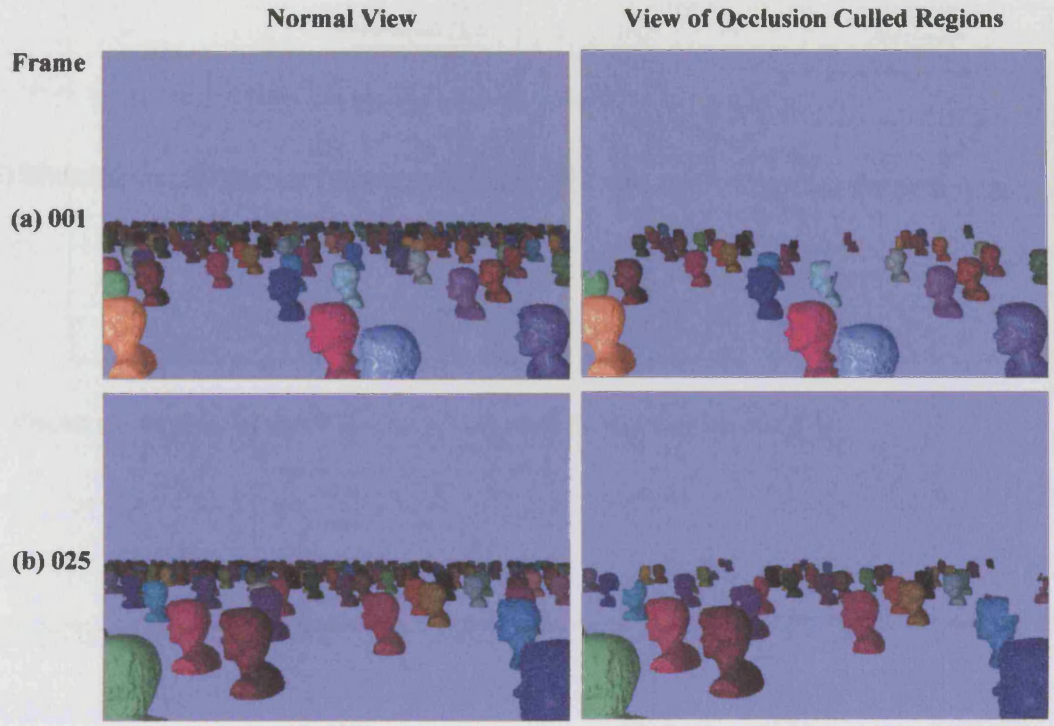


FIGURE 81. Higher views of frames 1 & 25, showing normal and occlusion culled results

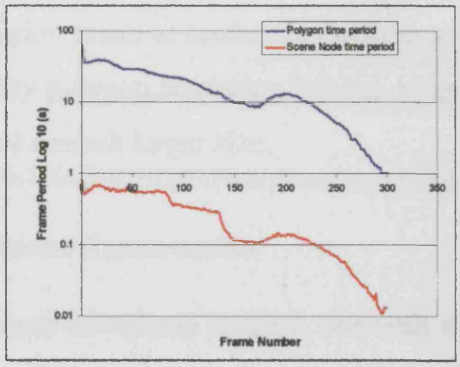


To visualize occlusion culled regions, nodes rendered from the viewpoints in example frames 1 and 25 in Figure 80 are shown from a higher viewpoint in the right hand column of Figure 81. A full, normal view of the scene at this higher viewpoint is shown in the left column for comparison. Comparing rendered images of the two systems for frames 1 and 25 in Figure 80, the output appears highly similar, but Figure 81 reveals substantial levels of occlusion culling.

Viewpoint collision detection has been configured with a scene node radius resolution of 0.02 as the threshold considered sufficient for triggering a collision. This is 25 times smaller than each object radius 0.5. The minimum distance setting is 0.1 from a node of this radius. Figure 85 shows the time periods and total nodes tested in each frame. The highest number of nodes tested in any one frame is about 80. The longest calculation period recorded was about 0.065ms which in itself could support a 15KHz refresh, so for this task, the collision detection is very practical and light weight.

FIGURE 82. (a) Frame periods (s) on Log 10 scale, (b) Scene node algorithm stage periods

(a)



(b)

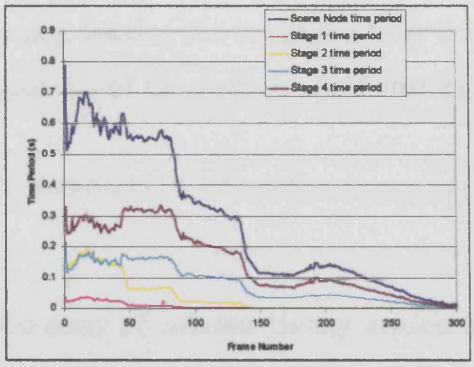


FIGURE 83. Number of nodes resulting from stage 3 (rasterized in stage 4)

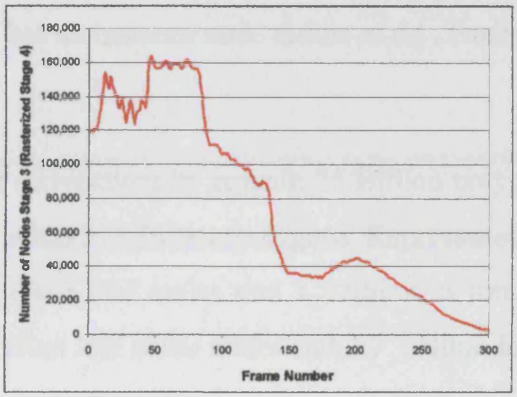
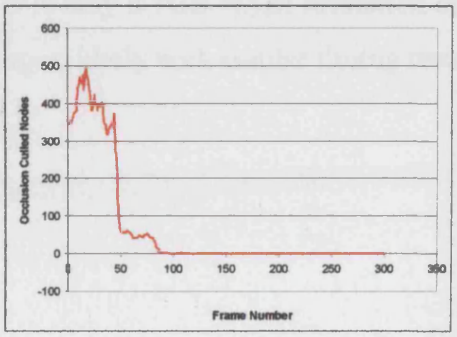


FIGURE 84. (a) Number of nodes occlusion culled, (b) mean view cone angle in degrees

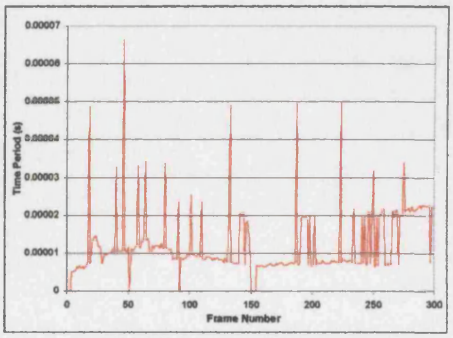
(a)



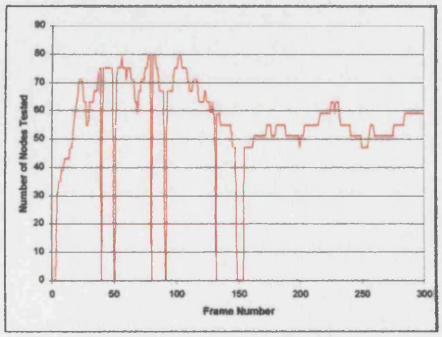
(b)

FIGURE 85. Viewpoint collision detection (a) time period, (b) total nodes tested

(a)



(b)



6.7 Massive Scene Rendering

This section looks at rendering massive scale scenes, leaving the baggage of the linear scalability polygon rendering behind, to get indications of the system's performance in scenes of a much larger size.

6.7.1 Scene Construction

Two scenes have been created, one with a jittered array of $600x600$ Bunny models and another with $600x600$ Female models. Each scene therefore contains $360,000$ models. A spacing of $1.2x1.2$ has been used, with a random offset of ± 0.5 for each ground plane component. Each model has an instance node radius of 0.5. Each model has random push down diffuse tinting.

The Bunny scene would conventionally contain 25 Billion polygons. The Female scene would conventionally contain 218 Billion polygons. Represented using scene nodes, the Bunny scene has 505 Billion leaf nodes and 1 Trillion in total in its scene tree. The Female scene has 1.3 Trillion leaf scene nodes and 2.7 Trillion in total in its scene tree.

These are massive scenes, the like of which are rarely seen in visualizations or games. In reality, it is only a 1GB RAM limitation on the test hardware that prevents these values being larger, likely with similar timing results.

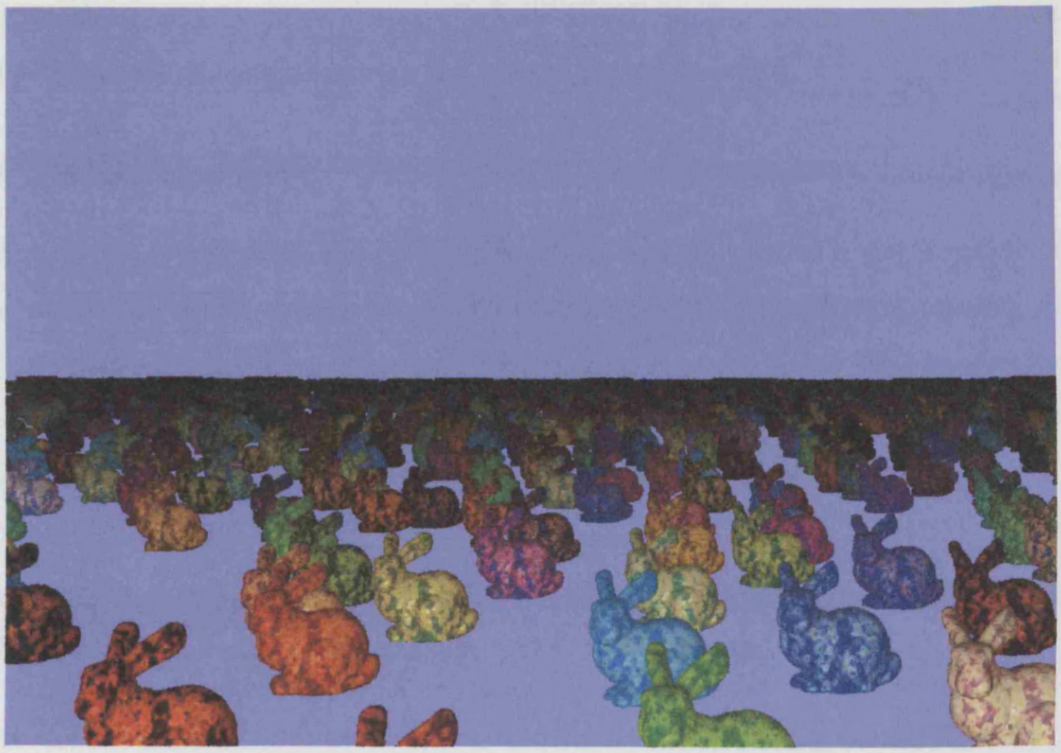
6.7.2 Results

One frame from each has been rendered, shown in Figure 86, from which the timings in Table 16 have been taken. Rendering parameters used are $t_1 = 2.5° (73p)$, $t_2 = 1.0° (29p)$, $t_{OccCull} = 1.9$ with $t_3 = 0.08° (2.3p)$. The viewpoint is placed beyond the center of one edge of the grid, above the models. Because occlusion is low, this is primarily a test of the level of detail control. The results are only just interactive at just over 2 seconds per frame, but this level of performance is dramatically improved in comparison to conventional rendering methods. A variation of about 5% has been measured between different frames.

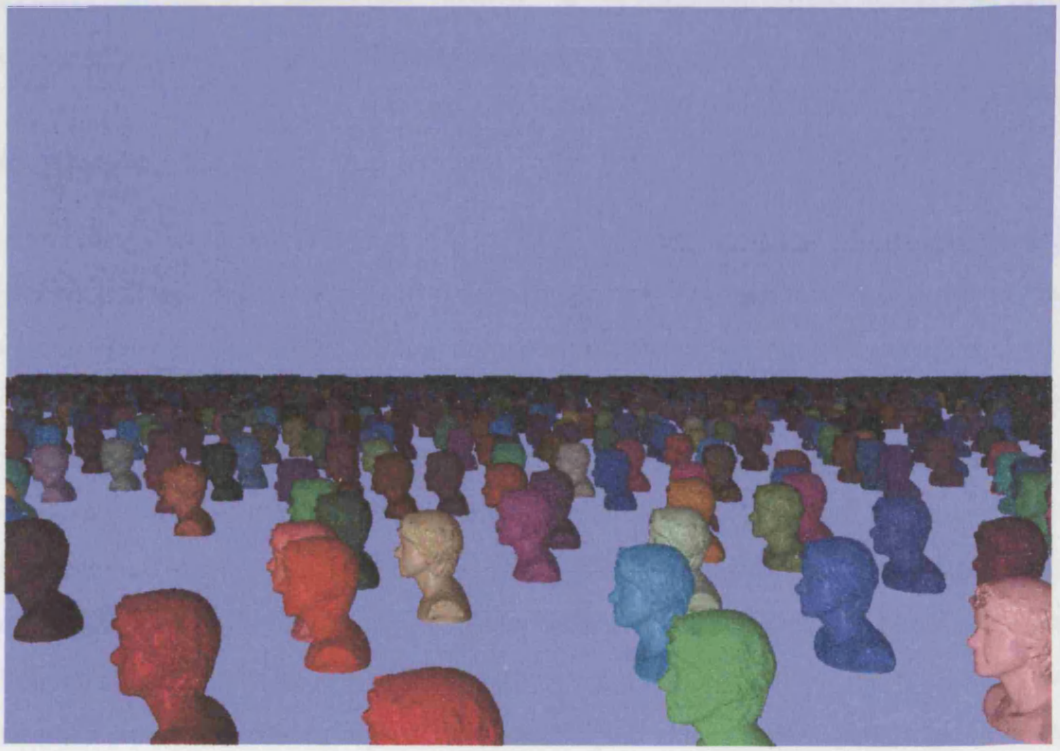
TABLE 16. Single frame performance details for Bunny and Female scenes

	Bunny Scene	Female Scene
Polygons / Leaf Scene Nodes	25Bn / 505Bn	218Bn / 1.3Tr
Time period frame (Total) (s)	2.213	2.260
Time period stage 1 (Setup) (s)	0.291	0.289
Time period stage 2 (Occlusion Culling) (s)	0.663	0.698
Time period stage 3 (Refinement) (s)	0.415	0.407
Time period stage 4 (Rasterization) (s)	0.843	0.866
Occluded Scene Nodes (s)	2,412	2,824
Number of nodes stage 4 (rasterized) (s)	403,180	428,518

FIGURE 86. Massive scene rendering of Bunny and Female scenes with 360,000 models each



(a) 360,000 Bunny models, 25 Billion polygons / 505 Billion scene nodes



(b) 360,000 Female models, 218 Billion polygons, 1.3 Trillion scene nodes

6.8 Densely Occluded Scene

This section examines the rendering of a very densely packed scene, comparing rendering performance of the system with and without occlusion culling.

6.8.1 Scene Construction

This is both a test of occlusion culling correctness and speedup, of a very dense scene. The scene consists of a dense block of $8x8x8$ staggered Bunny models, totalling 512 objects with 35M polygons. The same viewpoint is rendered firstly with image graph based occlusion culling and secondly, with no occlusion culling, using a simpler depth first refinement that replaces stages one to three. Rendering parameters used are $t_1 = 3°\,(88p)$, $t_2 = 1.5°\,(44p)$, $t_3 = 0.05°\,(1.5p)$, $t_{OccCull} = 1.9$ with a 1280x1024 window. An occlusion strength clipping value of 0.2 is also used to increase correctness around silhouettes, but potentially increasing the number of nodes rendered.

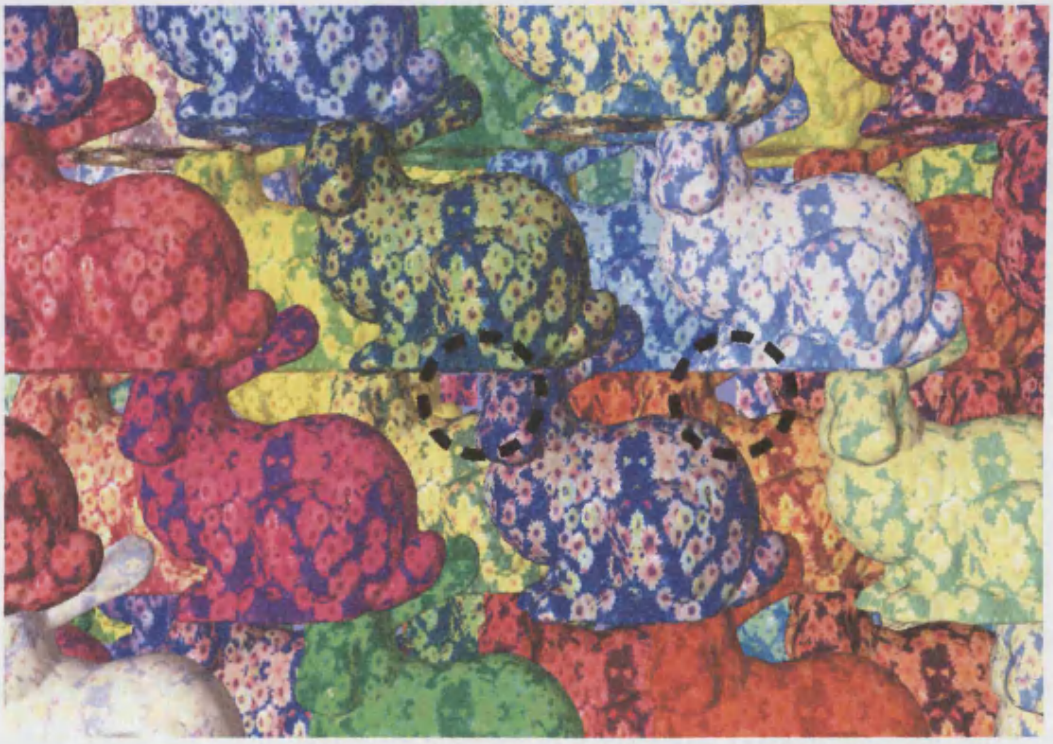
This scene is particularly costly to render without occlusion culling, because of the high degree of image coverage of a number of depth layers, rendered at higher level of detail with close packing. It is particularly difficult for the occlusion culling to correctly establish and cull regions, especially around silhouettes with vistas through objects.

6.8.2 Results

The resulting images are shown in Figure 87(a) with no occlusion culling and (b) with occlusion culling. The render period for the images are 53.3s and 15.7s respectively, representing about a 3.4x speedup when using occlusion culling. Note the small occlusion errors ringed in image (b). This is a typical form of error that can be encountered with more complex scenes. Although this particular example does not achieve interactive frame rates, it demonstrates a speedup due to occlusion culling that can be comparable to some results of other systems such as the Hierarchical Occlusion Map by Zhang and colleagues [228]. This is particularly notable because surfaces that are being occlusion culled would otherwise be rendered using level of detail control, which is not the case most other occlusion culling algorithms. Secondly, it is a purely software based occlusion culling technique that does not rely on hardware support.

FIGURE 87. Dense Bunny scene with 512 models (a) without OC, (b) with OC

(a) No Occlusion Culling, Render Time = 53.3s



(b) Occlusion Culling, Render Time = 15.7s (Small errors ringed)

6.9 Depth Ordering

The system is capable of providing a back to front depth ordering for rasterization, at the level of detail of the occlusion mask resulting from rendering stage two (see Section 3.3.2) Further refinement to rasterization levels of detail has arbitrary ordering that is maintained between frames. The approach is suitable for scenes where the depth complexity is typically one at this scene scale. This section will briefly examine some early examples, comparing these results with and without a hardware z-buffer.

6.9.1 Scene Construction

The images in Figure 88 and Figure 89 are rendered with parameters $t_1 = 2.5° (73p)$, $t_2 = 1.0° (29p)$, $t_3 = 0.08° (2.3p)$, $t_{OccCull} = 1.9$. Images (a) in both figures show the scene rendered using the occlusion mask ordering, configured for interactive frame rates at $t_2 = 1.0° (29p)$. No Z-buffering is used in these images. Images (b) in both figures show the scene rendered using a conventional hardware z-buffer for comparison. The first scene in Figure 88 is a simple test with two objects. At the occlusion mask level, a sort has occurred between image nodes until the two surfaces are deemed *separate_3d*, at some intermediate resolution, at which point, this status is maintained through the inheritance of this state between child scene tree image graph relations. The second scene in Figure 89 shows a jittered grid distribution of Bunny models with a grid spacing of 1.5 in both ground plane components, similar to that previously seen in Section 6.6 with the Female model.

6.9.2 Results

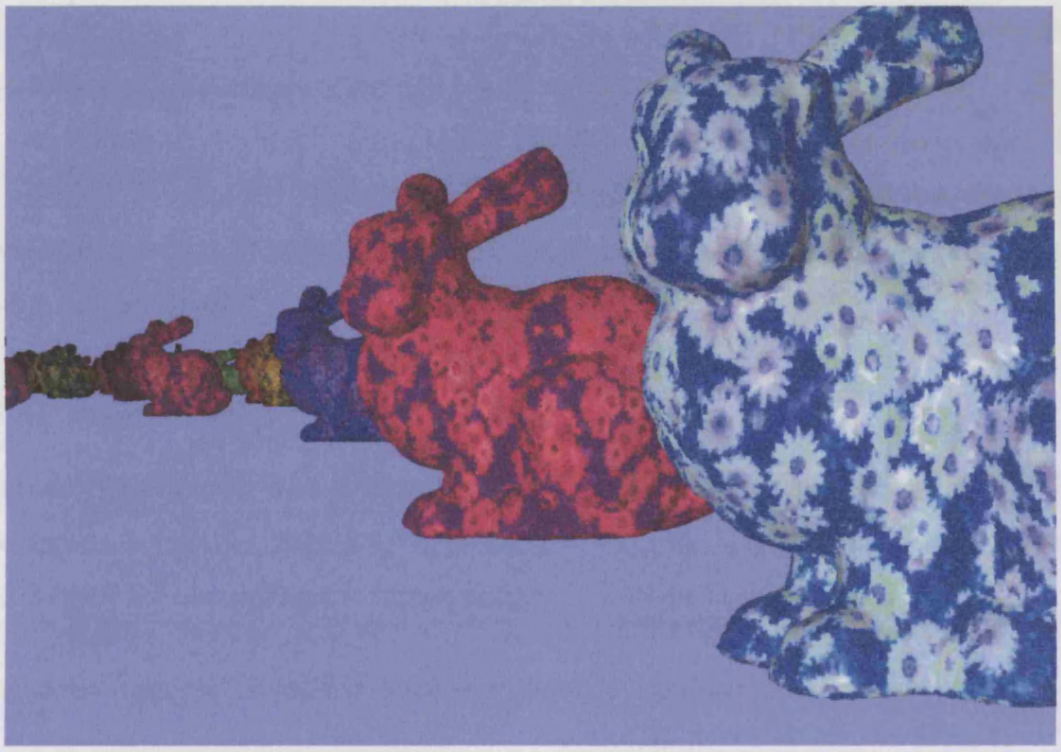
Comparing images (a) without z-buffer rendering and (b) with z-buffer rendering, the method appears to offer levels of error that are likely to be usable for some types of non critical applications and scenes where depth complexity is 1 at occlusion mask resolution for most views. Direct comparison of images is clearer when switching between them than studying them in printed form. Slight noise is visible in the occlusion mask ordered images, where the image nodes are rendered in a repeatable random ordering. Interactive walkthroughs of these scenes do not appear to display temporal noise between frames, even when approaching objects where level of detail switching will occur.

FIGURE 88. **Bunny and Venus models with occlusion mask vs. z-buffer depth ordering**

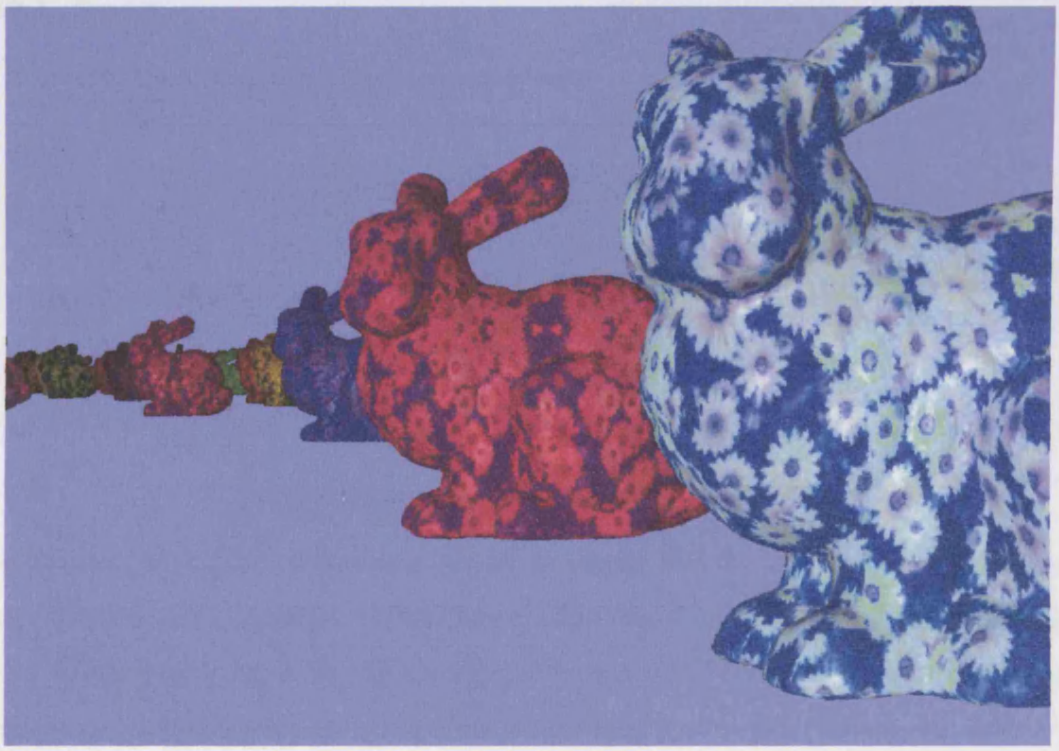
(a) Occlusion mask ordering, without z-buffer

(b) With z-buffer

FIGURE 89. Scene with 900 Bunny models with occlusion mask and z-buffer depth ordering



(a) Occlusion mask ordering, without z-buffer



(b) With z-buffer

6.10 Collision Detection

This section tests vertex-object collisions with two scenes for viewpoint collision detection. Object-object collision detection is not yet implemented, but the results are likely to be very similar to those of Quinlan [162]. The first is a single object to demonstrate the algorithm's performance characteristics in this basic case. In the second test, the viewpoint navigates around a scene with 900 Bunny models. Performance characteristics are shown for both cases.

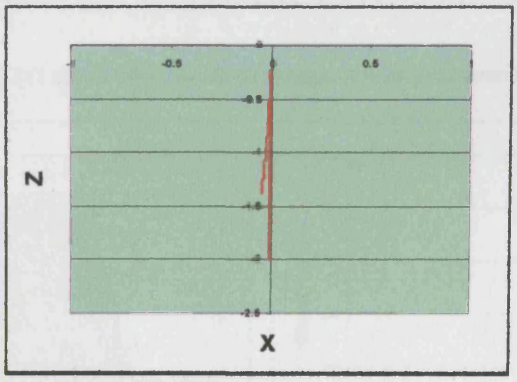
6.10.1 Single Model

A single Bunny model with instance bounding sphere radius 0.5 is positioned at [0 0 0]. The camera begins at a distance of 2, pointing towards the bunny, then moves towards it and collides for several frames. It then retracts. This sequence lasts for 73 frames.

Viewpoint collision detection thresholds have a minimum resolution threshold of $T_r = 0.02$ and a distance threshold of $T_d = 0.1$ in both tests.

6.10.2 Results

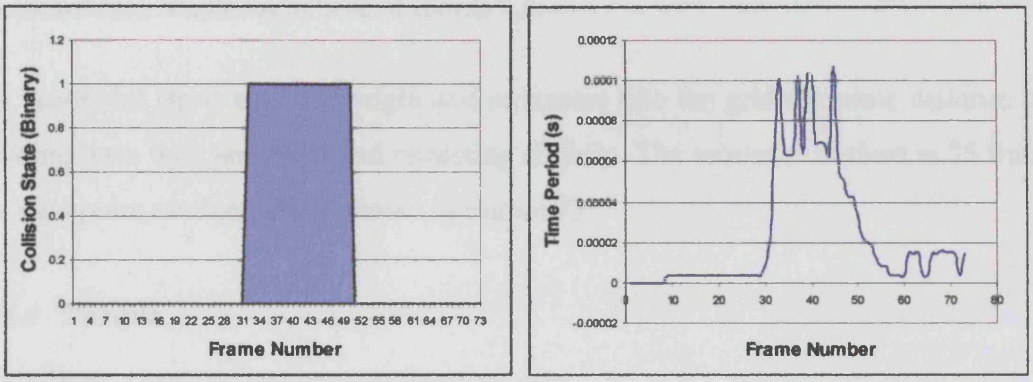
FIGURE 90. Single model fly-through viewpoint path



The motion path of the viewpoint is shown in Figure 90. The detected binary collision state is shown in Figure 91(a), where the collision begins at frame 32 and ends at frame 50. Collision time periods for the algorithm are shown in Figure 91(b), measured once. Measurements at this scale are susceptible to operating system fluctuations, but these figures give an indication of the scale of performance. As the viewpoint approaches the object to collide at frame 32, the overheads are extremely low, but quickly peaks near

collision to about 0.1ms. Throughout the collision period where the viewpoint attempts to move through the object but is prohibited, the overheads remain peaked but fall off as the camera starts to retract near frame 50.

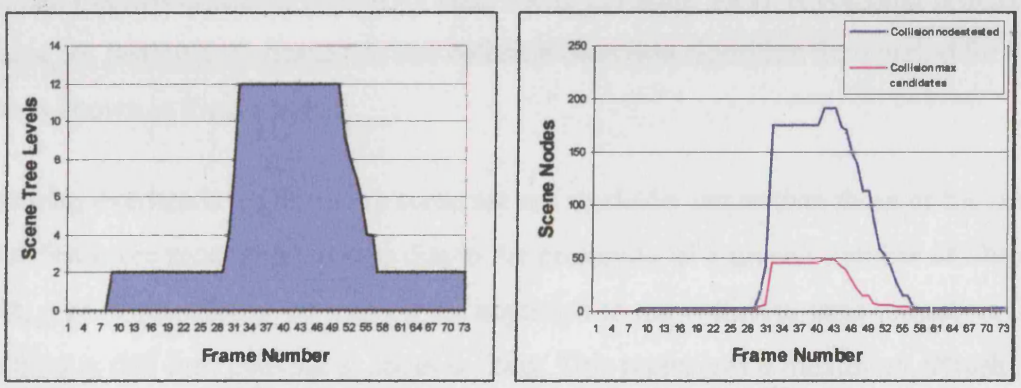
FIGURE 91. Single model (a) binary collision state, (b) viewpoint collision detection time period (s)



(a) (b)

The number of scene tree levels accessed to calculate the collision detection is given in Figure 92(a). This stays low at just two scene tree levels until peaking when approaching the object to 12 levels. The maximum number of collision candidates at any tree level during processing is shown in Figure 92(b), along with the total number of candidate nodes processed. Again, the peak is clearly visible from approach to retraction, but processing overheads remain very low, peaking at under 200 node tests per frame.

FIGURE 92. Single model (a) scene tree levels (b) scene nodes processed and max level candidates



(a) (b)

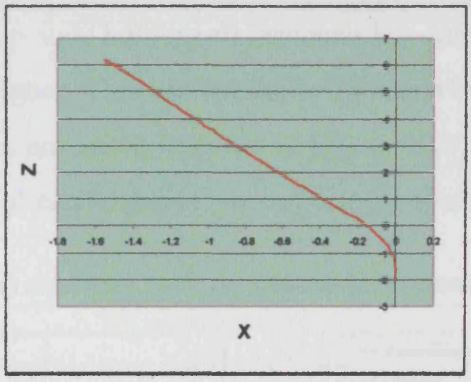
6.10.3 Large Scene

Having seen the performance characteristics for a scene with a single object, this section will look at the characteristics for a more complex scene with 900 Bunny models in a randomized jittered grid with a spacing of 1.5 and a random offset of +/-0.5. Each model has an instance bounding sphere of radius 0.5.

The viewpoint starts near the origin and navigates into the grid for some distance until colliding once with an object and retracting slightly. The sequence is short at 75 frames. The viewpoint motion path is shown in Figure 93.

6.10.4 Results

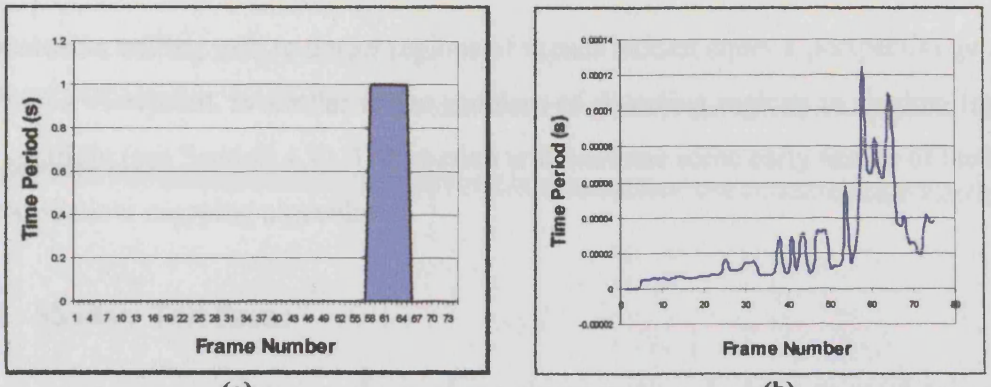
FIGURE 93. **Large scene fly-through viewpoint path**



The binary collision state is shown for each frame in Figure 94(a). A collision is detected at frame 59, lasting until frame 65. The collision detection algorithm time period for each frame is shown in Figure 94(b).

Processing overheads for this large scene are not markedly larger than those of the single model, but more processing occurs due to the proximity of a greater number of objects. Again, a peak can clearly be seen on the approach to the collision, until retraction. The algorithm is still fast, peaking at about 0.12ms. This represents a maximum refresh rate of about 8 KHz.

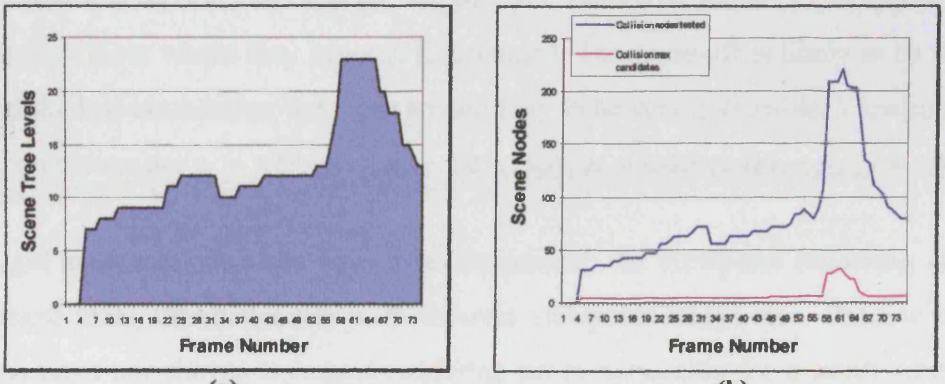
FIGURE 94. Large scene (a) binary collision state, (b) viewpoint collision detection time period (s)



(a) (b)

Because the scene is more complex, the algorithm processes more tree levels when navigating, shown in Figure 95(a). The general background processing level is at around 10 scene tree levels before all collision candidates are rejected. Because the scene contains multiple objects, the group node holding all instances has constructed a partitioned hierarchy, making the tree higher. This case peaks at 23 scene tree levels during collision. The maximum total nodes processed is shown in Figure 95(b), which still peaks at a low 230. The largest number of candidates at any one tree level is low, peaking at about 30.

FIGURE 95. Large scene (a) scene tree levels (b) scene nodes processed and max level candidates



(a) (b)

6.11 Shadows

The occlusion culling task to detect regions of scenes hidden under a perspective projection from a viewpoint, is similar to the problem of detecting regions in shadow from a point spotlight (see Section 4.1). This section will examine some early results of the hierarchical shadow mapping algorithm.

6.11.1 Shadow Test Scene

A basic scene is used to test the affects of one object casting shadows on another in close proximity. A single light source shines from the left, illuminating the front Bunny model, which casts a shadow on the rear Bunny with a green push down attribute diffuse colour tinting. The shading for this test is performed in software rather than using OpenGL shading to allow the influence of lights to be modulated based on whether image nodes are in illuminated or in shadow.

Light source rendering parameters used in these examples are $t_1 = 2.5° (73p)$, $t_{OccCull} = 1.3$ with variable occlusion mask resolution t_2. Rasterization is not required for shadows, so the t_3 threshold is irrelevant. A low occlusion value has been chosen to ensure shadow detection after a single depth layer, otherwise artifacts can appear in the shadowed surfaces where they become illuminated. The trade-off is likely to be slightly enlarged shadow boundaries, but these are unlikely to be very noticeable. Viewpoint rendering parameters are $t_1 = 2.5° (73p)$, $t_2 = 1.0° (29p)$, $t_3 = 0.08° (2.3p)$, $t_{OccCull} = 1.9$.

Once light source image trees have been calculated, the viewpoint rendering shadow traces these trees simultaneously with its own viewpoint image tree. Shadow tracing functions have not shown to impact rendering times noticeably for a small number of light sources.

6.11.2 Shadow Level of Detail

Shadow generation using the occlusion graph system requires more processing for light source image graphs than for viewpoints. This is because t_2 refinement levels must be higher resolution to provide a sufficient boundary to the shadow. This level of detail is

selectable and may be varied based on speed and quality trade-offs, or could be reduced during rendering of dynamic scenes when not static.

To demonstrate the affects of level of detail in shadow calculation, three occlusion mask thresholds t_2 are tested with values $[0.8, 0.4, 0.2]$.

6.11.3 Results

Images for each t_2 value are shown in Figure 96. The calculation periods to generate the each light source image graph are shown in Table 17, with the number of nodes in the occlusion mask. Additional shadow tracing overhead to render the viewpoint is negligible. Virtual memory is required for these tests that require more than 1GB once all operating system and application overheads are added with this scene. As such, they should only be taken as a very approximate guide and are likely to be faster when run without virtual memory. Once shadows have been generated, they need not be recalculated until either the light source or objects move. The results show performance times that may be suitable for some applications, but the current version of the method is not currently suitable when implemented purely in software. Future work may enhance the speed of occlusion culling, e.g. using hardware support and frame coherence to make its use more feasible. The appearance of the shadow boundaries clearly sharpens with increasing level of detail from (a) to (c). The slight blocky appearance at the boundaries is attributable to the spatial partitioning system used to construct the objects after sampling. When the light source image graph is shadow traced, the viewpoint's refinement goes beyond the LOD of the shadow map image graph, where the illumination state is inherited by higher LODs. These hierarchical partitions are object axis aligned on the largest axis, forming rectangular volumes that contain a number of nodes at higher resolutions. Future work could be undertaken to reduce the appearance of these forms of artifacts using other partitioning schemes or boundary softening methods.

TABLE 17. Shadows at multiple levels of detail

Occlusion Mask Threshold t_2 View Cone Semi-Angle (Degrees)	Single Light Calculation Time Period (s)	Scene Nodes in Occlusion Mask
0.8	0.309	3,558
0.4	0.480	7,361
0.2	1.335	25,800

FIGURE 96. Shadows with increasing light source image graph level of detail

t_2

(a) 0.8

(b) 0.4

(c) 0.2

6.12 Comparison with Other Systems

This section will compare the different aspects of the results presented here with a selection of relevant systems detailed in Chapter 2, particularly PBR systems in Section 2.9.

6.12.1 Scene Sampling and Storage

Pointer use is a concern in both Canopy, QSplat [171] [172] and similar hierarchical scene representation systems. QSplat uses a merging process to reduce their binary hierarchy to a branching factor stated to be close to 4. The pointer system used in QSplat is significantly different. Each set of nodes that are children of the same parent in the tree level above, have a single pointer that refers to all their children in the next tree level below. This encoding scheme is reliant on a literal breadth first storage. No pointers are included if no child nodes are present.

Given a tree system with B branching nodes and L leaf nodes with a strict or mean branching factor F, certain relationships will hold:

$$L = 1 + B(F-1) \qquad \text{(EQ 59)}$$

and therefore:

$$B = \frac{L-1}{F-1} \qquad \text{(EQ 60)}$$

(EQ 60) tends towards $L/(F-1)$ with increasing L. QSplat has one pointer per branching tree node. If the QSplat system realizes a mean branch factor of 3.5 as stated, the number of branching nodes and therefore pointers is given by (EQ 60) based on the known number of source leaf nodes $B = L/2.5 = 0.4L$.

The container hierarchy system has pointer to leaf node ratios of $0.74, 0.73, 0.6, 0.74$ for the Bunny, Venus, Isis and Female models respectively. These values disregard the fact that an additional virtual function pointer is required in each container. In the case where this could be argued as a requirement for the container system, regardless of the additional flexibility it gives the system, if one pointer is added for each container, these ratios become $0.84, 0.84, 0.98, 0.83$ respectively as coefficients of the number of leaf nodes. Currently, this is slightly less efficient than QSplat's 0.4, but comparison is

harder because the scene trees within the containers in a scene graph have an integer binary tree branching factor of 2, which offers nearly twice as many levels and therefore levels of detail than QSplat, with less than twice as many pointers to achieve this and greater potential for extension. QSplat also uses an additional 2 bits per node to encode the number of children. It is expected that the addition of skew containers and containers with specific child pointers using template programming will improve the efficiency substantially, although skew containers will substantially increase the complexity of the packing algorithm due to the size of the search space.

Examples of the Surfels system by Pfister and colleagues [158] show model file size storage requirements as having a mean of about 25,200 surfels per MB when using their 3 to 1 reduction system. The example models in this section have a mean of 75,485 total scene nodes per MB. This comparison is only rough, but indicative of over all efficiency.

Sampling times are of practical importance when pre-processing and of great importance if ever integrated into run-time procedural reconstruction systems. Voxelization times are low, ranging from about 2 seconds for the Bunny, to 7 seconds for the Female model, realizing a sampling speed of over 1M voxels per second. Hierarchical construction, including spatial partitioning, attribute calculation and container packing for the models operates at an average of about 1s for every 70,000 input leaf scene nodes. Total voxelization and construction times range from about 18s for the Bunny to 60s for Female.

The sample performance figures compare reasonably with those presented in the most similar system, Qsplat by Rusinkiewicz and Levoy [171] [172], considering that Canopy has additional attribute overheads, double the number of LODs and container packing. Canopy also voxelizes polygons rather than sampling vertex patches, which will be more computationally expensive. In their system, using a simple rule of thumb measure, the model pre-processing averages at a processing speed of about 50,000 samples per second. It must be noted that the hardware used for these performance tests is significantly more powerful than that used in QSplat's tests, which is stated by be an Onyx2 with unspecified processor and memory resources. It is likely however, that due to the added costs of container packing, this process may be slightly slower than that of QSplat.

The Surfels system by Pfister and colleagues [158] is reported to take about one hour to sample their example models at 6 LOD surfel mipmaps with samples numbering less than one million per model. Canopy's voxelization technique is substantially faster by about three orders of magnitude. Including hierarchical construction, the Canopy system is at least one order of magnitude faster, though comparison is difficult, particularly as the Surfels system derives superior quality surface sampling with anti-aliasing.

The Layered Point Clouds system by Gobbetti and Marton [72], like many point based systems, assumes that the surface is specified as a point set, without sampling. The out of core construction times however, are notably fast at just over 2 mins to build a point cloud hierarchy for a 4M point model, that is marginally slower than achieved by QSplat and Canopy.

Wu, Zhang and Kobbelt [220] compare a number of point based simplification schemes, including their own, where the majority of systems including Pajarola [152], Pauly Gross and Kobbelt [154] and Wu and Kobbelt [219]. The first of these three construct LODs of point based models in less than 6 mins for a model with 600k points. Only the latter required more time at about 13 mins. Although substantially slower, the quality of LODs in these systems is likely to be higher than that of Canopy or QSplat.

Comparing more broadly, polygonal simplification systems for level of detail model provision have themselves reported widely differing processing times. The progressive mesh system by Hoppe [100] [99] [102] has reported processing times as large as 10 hours for a model of the grand canyon model with 360,000 vertices and 717,000 polygons [102]. The Stanford bunny model used in this chapter is also quoted as requiring a pre-processing time of 51 minutes. The processor used is quoted at 150MHz which is comparatively slow against a 2.8MHz system used to obtain the figures in this chapter, but the speedup that would be achieved by using a faster processor is unlikely to be as much as a linear multiple suggested by the clock speed differences, primarily because system memory speeds have not matured at the same rate. Hardware differences accepted, the sampling and hierarchical reconstruction is still likely to compare favourably time wise, although the progressive mesh method is likely to result in better quality level of detail approximations. However, at near or sub pixel levels, differences are not likely to be noticeable in images.

The vertex clustering method by Luebke and Erikson [136] is reported to require a pre-processing time of 158 seconds for a CAD model with 412,000 vertices, 699,000 triangles. Again, even though hardware used will differ significantly, the Canopy and QSplat methods have comparable times.

Lindstrom and Turk [133] compare a number of polygonal simplification methods tested with the bunny model in this chapter. To generate a range of models at various reduced resolutions, the algorithms discussed have processing times ranging between several minutes and over two hours for a sequence of discrete models. Again, although hardware differs, our method appears to be at least comparable.

6.12.2 Rendering Performance

Direct comparison with other point based systems is difficult due to the varying types of scalability, scene design, viewpoints, scene sizes, image qualities, hardware used, image size, extensibility and levels of implementation optimization being just a few reasons. In particular, systems that do not incorporate occlusion culling may be advantaged due to lack of overheads when rendering scenes with low occlusion. However, a few approximate comparisons will be made with a selection of notable and scalable point based systems.

QSplat by Rusinkiewicz and Levoy [171] [172] incorporates level of detail, but not occlusion culling and tests are given for single objects only. Frame periods for a $1280x1024$ window for their *static* rendering are about 0.8 s on an SGI Onyx2 Infinite Reality. Equipment differences considered, QSplat is likely to run several times faster, but lack of a scene graph architecture and occlusion culling makes comparison less valid. QSplat is likely to be slower for large environments with dense occlusion.

Similarly, the Surfels system by Pfister and colleagues [158] has level of detail but no scene graph architecture or occlusion culling and tests are only given for single objects. With a $1024x1024$ image, frame periods are about 1 s on a 700MHz system, but image quality due to anti-aliasing is higher. Generally, this system is faster with single models, but does not attempt to occlusion cull.

Coconu and Hege [36] render grids of objects using their system based on an octree of LDIs for LOD and higher quality EWA splatting and hybrid rendering. A scene with a grid of $200x200$ Pisa models that would comprise 1.4×10^{14} polygons is rendered with a 2 pixel splat size. Frame periods of about 0.76 s are achieved with an image size of $500x400$ on a 2GHz system. With a splat size of 3 pixels and Advanced Graphics Port (AGP) based point caching, frame periods drop to about 0.17 s, highlighting the speedups that can be obtained through low level optimization. For comparison, the image area is $1/4$ of the $1280x1024$ used in our tests and so as an estimate, periods could be $x4$, becoming about 3 s and 0.68 s respectively. Tests of our own system without GPU level caching optimizations would indicate comparable speeds for similar scenes against their non-AGP cached results. To compare the effects of occlusion culling, a more densely occluded environment is required.

Comparing Canopy with the Randomized Z-Buffer by Wand and colleagues [205], on more minimal hardware, the authors achieve frame periods of about 4.6 s for a scene of 10^{14} polygons and an image size of $640x480$. Canopy's frame period with 2.8×10^{11} polygons and a $1280x1024$ image on more powerful hardware is about 2.2 s, with slightly larger splat sizes. Scenes and viewpoints are different and this can have substantial impact on frame periods, as shown in Section 6.6. Canopy requires an order of magnitude less pixel over-draw. Multiplying up for image size as a simple estimation, the randomized z-buffer's period is $4 \times 4.6 = 18.4$ s and making a probably conservative estimate of 2.8GHz/0.8GHz = 3.5 speedup for equipment difference, leaves a period of $18.4/3.5 = 5.3$ s. Over all, the performance of the two approaches appear to be approximately comparable when caching is not used. GPU based caching in Canopy would need to be used to compare caching speedups. Again, comparison with a more densely occluded environment would be required to demonstrate efficiencies of occlusion culling in the canopy system.

The Deferred Splatting system by Guennebaud, Barthe and Paulin [81] is shown to render 6,800 point based tree models, with $750k$ points per tree at about 11 fps on equivalent hardware and an image size of $512x512$, using LOD and a hardware based occlusion culling process. Again, as an estimate, multiplying the frame period by four to equal

image sizes, gives $4/11 = 0.36$ s. This technique is generally faster, by several times, though it relies on hardware and frame coherence to carry out occlusion culling and is substantially optimized. The system also has higher quality splatting.

The Layered Point Cloud system by Gobbetti and Marton [72] again uses hardware occlusion culling with frame coherence. Tests presented differ, in that a single large iso-surface dataset is presented. With a 335x335 window, the authors render about 234M point object with a frame period of about 0.18 s with single pixel tolerance on equivalent hardware. About 6.3M splats are actually rasterized. This window area is more than $10x$ less than $1280x1024$, although in our tests, not all image space is addressed. The level of overdraw is still markedly high. More similar scenes, particularly with greater depth complexity would be required to compare these systems further.

The Sequential Point Trees system by Dachsbacher, Vogelgsang and Stamminger [48] contrasts our system and others by caching point hierarchies in GPU memory, where throughput is far higher due to specific design for this style of linear throughput. Frame periods of just 0.011 s and 0.027 s are obtained for a scene consisting of four well known scanned models and several trees. Notably, this technique is not able to hierarchically cull against the view volume. Such caching approaches may complement CPU based approaches, including the Canopy system.

Comparing against polygon based systems with level of detail and occlusion culling, the Gigawalk system by Baxter and colleagues [14] renders a scene with 13M polygons and $1,200$ objects at an average frame period of 0.05 s on a 300MHz CPU and an SGI Infinite Reality 2 graphics system. Yoon, Salomon and Manocha [224] render a similar scene with 12M polygons and $1,200$ objects on nearly identical hardware to that used in this chapter at frame periods of about 0.08 s. These test scenes are somewhat smaller than those presented here or point based systems [81] and [205], although the scenes are potentially more representative of real world use. The system has high frame rates and is likely to scale similarly. Again, this system relies on hardware based occlusion culling and heavier pre-processing may inhibit application to dynamic scenes.

Andujar and colleagues [9] integrate a level of detail controller with occlusion culling. In a scene with 3,903 objects and 225,775 polygons, with 3 LODs per object, frame periods of 0.323 s and under are achieved on a 194Mhz CPU system. These scenes are relatively small, so useful comparison is hard. The approach appears to be fast, although it is not readily applicable to dynamic scenes due to pre-processing.

The most common technique for real-time shadows is shadow mapping [214] [58], which suffers from the same linear scalability as standard rendering algorithms and the standard pipeline. Due to the speed of modern graphics cards, this technique can work well with smaller scenes at real-time frame rates. Image graph based hierarchical shadow mapping is substantially slower with smaller scenes, firstly due to being a software based approach and secondly, due to the requirement to refine the graph to higher levels of detail to achieve sharper shadow boundaries. For very complex scenes however, the image graph system should benefit from sub-linear complexity, although we do not present examples here.

6.13 Summary

This chapter tested scene sampling and scene tree construction, giving performance details for voxelization and hierarchical scene tree construction, finding this task to be comparable with similar systems in terms of storage and time required.

Level of detail examples were then given with varying view cone angle thresholds for refinement in stage three, to visualize rendered levels of the scene tree.

Occlusion culling was tested with several culling thresholds, one that shows no artifacts, one that shows substantial artifacts and one that does not trigger any culling.

Scalability of depth complexity was shown for models over distance, comparing against standard polygon rendering with no LOD or occlusion culling. The standard method shows strong linear scalability, whilst the algorithm shows sub-linear scalability. Occluder fusion behaviour was also demonstrated.

A large scene with 900 models consisting of about 544M polygons was rendered as a walkthrough using the algorithm and a standard polygon rendering system with no LOD or occlusion culling. Occlusion culling was visualized for cases of this walkthrough. Performance times for the algorithm were a mean factor of approximately 70 times faster.

To test the algorithm's scalability on its own, two massive scenes were rendered, with a grid of 600x600 Bunny models and another with 600x600 Female models. These scenes would conventionally consist of 25 Billion and 218 Billion polygons respectively. Represented as scene nodes, the scenes have 505 Billion and 1.3 Trillion scene nodes at leaf level and 1 Trillion and 2.7 Trillion in total for each scene. Rendering was shown to scale sub-linearly, primarily due to LOD, with rendering periods of just over 2 seconds on the moderate test hardware.

Performance with and without occlusion culling was evaluated for a densely packed scene with high and complex occlusion properties, using a simple refinement algorithm for non occlusion culled rendering. A speed-up of more than 3x was observed, with small but notable errors in resulting images.

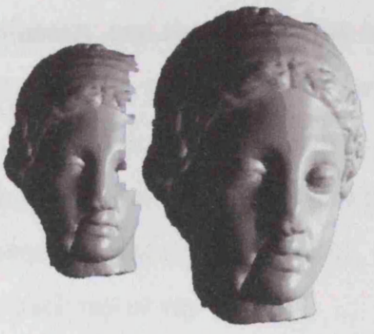
Approximate back to front depth ordering for rasterization was then tested with a simple scene consisting of two objects and a more complex scene comprised of a grid of 900 Bunny models. Reasonable quality rendering was achieved for these two scenes at these scales, but more testing is required with other types of scenes at various scales to establish the degree of usefulness of the technique.

Viewpoint-object collision detection was then tested with a single object and then a walkthrough for a single set of parameters. Results show the overheads to be very small, with peaks at just 0.1ms.

A simple, early example of image tree shadow tracing was given with timing results for image graph refinement for one point light source. Once generated, future renderings do not require recalculation of the light source image tree unless the light or scene moves. Results show frame periods from 0.3 s to 1.3 s for this CPU based algorithm with a small scene, but are currently about two orders of magnitude slower than hardware shadow mapping implementations. With larger scenes, the algorithm will fair better due to its sub-linear scalability against linear scalability.

The results were very approximately compared with other systems, concluding that the algorithm is often at least comparable other techniques, but further tests with greater commonality in scene types and increased depth complexity would be required for more accurate comparison.

Chapter 7 *Conclusion*



This chapter concludes the thesis, examining ***contributions*** in Section 7.1.

This is followed by ideas for ***future work*** in Section 7.2.

Final ***concluding comments*** are given in Section 7.3.

7.1 Contributions

Standard rendering algorithms and hardware pipelines scale linearly with an algorithm complexity dependent on the size and detail of the scene. Various standard techniques have been developed to achieve sub-linear (better than linear) scalability in specific areas, but few have been developed that combine multiple solutions and even fewer that tightly couple these solutions to unify data structures and share processes.

This thesis has presented a point based rendering algorithm that is a unified solution to multiple sub-linear scalability problems, addressing many of the main topics covered in Chapter 2 (see Summary in Section 2.10).

Although the upper bound is $O(j^2)$ in the number of scene leaf nodes j, in practice the algorithm scales better than linearly and the worst case is very unlikely to be encountered. The algorithm is output sensitive, such that it can be independent of the maximum level of detail present in the scene and due to surface aggregating level of detail, can also be independent of the maximum number of depth layers in the view volume. To render an image, the scene tree is traversed just once per frame, unlike ray casting approaches that traverse the hierarchy for each ray or ray bundle.

With respect to each of the problems highlighted in Section 3.1, solutions have been integrated for *locale management, view volume culling, back face culling, occlusion culling, depth ordering* and *level of detail control*. Rendering termination is left as future work (see Section 3.3.8). These solutions operate on the same scene tree representation that unifies the concept of objects and primitives across scene scales.

A small number of systems combine LOD and occlusion culling in polygon based systems, including The Berkeley Walkthrough system by Funkhouser and colleagues [65], the MMR system by Aliaga and colleagues [7], Gigawalk by Baxter and colleagues [14], Andújar and colleagues' [9] Hardly Visible Sets (HVS), El-Sana, Sokolovsky and Silva [54], Yoon, Salomon and Manocha [224] and Govindaraju et al. [76]. Even fewer point based systems have incorporate occlusion culling, but include the Deferred Splatting system by Guennebaud, Barthe and Paulin [81], the Layered Point Cloud system by Gobbetti Marton [72] and the voxel based splatting hybrid Far Voxels by Gob-

betti and Marton [71]. Systems such as Rusinkiewicz and Levoy's QSplat [171], [172], Pfister and colleagues' Surfels [158] and the Randomized Z-Buffer by Wand and colleagues [205] notably do not occlusion cull.

To our knowledge, this is the first point based system to perform occlusion culling by directly exploiting the geometry of a point based hierarchy, without the use of hardware rasterization for occlusion tests. The only rendering data structures required at run-time are the *image graph* and optional *image tree*, in addition to scene graph data structures.

Whilst the architecture has so far only been applied to static scenes, it does not significantly prohibit extension to dynamic scenes and soft body dynamics, using lazy or strict rebuilds of regions of the scene tree.

Scene topology is not considered or maintained and objects and surface components are freely permitted to have intersection or containment relationships with each other in object space, unlike many systems based on spatial partitioning that require unique spatial occupancy to function. This may be particularly useful when composing a scene from multiple existing hierarchies representing objects or sub-scenes that may intersect. It also allows dynamic rigid body objects to be represented by scene trees as children of a group node, where only group node scene tree rebuilds are required.

Hierarchical surface colouration is used, in preference to applying texture maps. There is therefore no need for surface parameterization to achieve unique one to one mappings between texture space and surfaces for *texture baking* to cache surface shading results, which is commonly a prohibitively expensive task for very large, detailed scenes.

Level of detail approximations in the bounding sphere hierarchy are not necessarily tight, accurate representations in object space, but are effective when combined with image space metrics in a selective refinement algorithm. Hierarchical attribute approximations provide a form of object space anti-aliasing, similar to mip-maps in texture mapping.

The level of detail control algorithm is primarily concerned with image quality, taking a level of detail culling approach, rather than meeting a target frame rate or frame rate consistency requirements, but alternative control schemes can be integrated.

Parameters specify a trade off between time and quality. Near pixel levels of detail with a simple splatting system appear to offer reasonable image quality although the technique may benefit from image pre-filtering. We have not focused on high quality or high performance splatting. Graphics hardware is intentionally under-used in the implementation and similar results could likely be achieved were the rendering implemented purely in software. Adaptation for hardware support and anti-aliasing are left as future work.

The algorithm's occlusion culling is an approximate *from-point* technique with occluder fusion. It can be configured to be typically conservative in many cases such that all surface regions that should be visible are rendered, but the technique provides no general, absolute guarantee of conservative occlusion culling. In some circumstances such as viewing distant objects through a vista of successive silhouettes or through tightly curved silhouettes, some small errors are occasionally noticeable and future work is required to improve the accuracy of occlusion estimation.

Occlusion culling has been shown to provide a 3.4x speedup for a densely occluded scene, compared to a non image graph based refinement algorithm. Occlusion culling in combination with LOD, potentially culls surfaces that would otherwise be rendered at lower levels of detail and therefore, the advantages of occlusion culling are potentially reduced. The image graph approach has demonstrated speedups for highly occluded scenes, with speedup factors comparable with some results from systems that cull full resolution surfaces, including Zhang's Hierarchical Occlusion Map [228], but more similar scenes and walkthroughs are required for more accurate comparison.

Occlusion culling also permits the identification of a set of potentially visible surface regions that supports the image tree caching scheme and potential additions for GPU based caching of visible surfaces.

In addition, we have demonstrated the algorithm's applicability to shadow calculation and collision detection. Whilst collision detection remains fast and feasible, the image graph process applied to shadows is still practically slow compared to modern hardware shadow mapping techniques for small scenes, but as with viewpoint rendering, should benefit from sub-linear scalability for larger scenes.

We have shown how the data structures of the rendering system can be geometrically compressed in a way similar to QSplat [171] [172]. Whereas most point based systems are concerned with quality rendering of single objects, this research has concentrated on developing the algorithm within a scene graph architecture that will support large scenes. Scene tree regions of the scene graph also have pointer compression that substantially reduces hierarchy storage overheads in an extensible system for increasing efficiency.

The scene graph architecture's outward appearance should be familiar to most graphics developers, whilst hiding the existence of the spatially partitioned hierarchy and can be made transparent for algorithms that wish to simply traverse the scene tree. A basic framework for procedural scene generation is included, as is basic read on demand to support out of core rendering.

Results for a non-pipelined implementation without substantial code level optimization have shown significantly sub-linear rendering properties compared against linear rendering. Although the current implementation is significantly slower than hardware based polygon rendering for small scenes, it has shown to be much faster for very large, detailed scenes. For practical application, future work can concentrate on speedups to achieve realtime frame rates.

Performance is considered to be often at least approximately comparable to various other combined solutions and point based systems (see Section 6.12) although some hardware oriented systems are shown to achieve substantially faster frame rates with smaller scenes. Direct comparison is at best approximate in most cases and more precise speed comparison can only be established using equivalent scenes and walkthroughs.

7.2 Future Work

A large number of research topics are viable to enhance the algorithm, particularly to make it fully practical with respect to real-time frame rates and frame rate consistency.

Further work is required to achieve better scene tree construction, storage efficiency, locale management and improved occlusion culling accuracy. Although the system scales better than linearly, frame rates must be improved for practical use on current hardware. Rendering refinement termination is also required to end rendering as early as possible, when it can be detected that no more scene detail will contribute to the image. Further work is required to realize a fully automated scene graph system that solves rebuild requirements to maintain hierarchical consistency, when dynamic changes occur in the scene.

Alternative hierarchical construction algorithms are applicable, such as better BSP tree policies, PCA splitting approaches or other forms of point based surface optimization could be used (see Section 2.9.4) if higher quality LODs are required. The basic quality driven LOD control system could also be replaced with one based on meeting target frame rates or consistency, using techniques outlined in Section 2.3.5. An emphasis is placed here on quality, in the expectation that hardware power will deem them less necessary due to the ability to refine to pixel levels in all frames.

Greater storage efficiency could be achieved by implementing *skew* containers, that store scene tree sub-trees that are irregular shapes that are not full trees. Containers currently have a maximum number of child pointers or no child pointers for leaves. This may also be altered to create containers with specific numbers of child pointers that will remove NULL pointers in all branching containers.

Locale management in the algorithm is currently very simple and limited to static scenes. Future work may investigate locale management in dynamic scenes, where primary problems occur due to intersection and overlap possibilities in the scene tree.

The correctness of the rendering algorithm may be improved, particularly to achieve better occlusion culling functions, e.g. to penalize cases where the current summation of occlusion contributions may incorrectly trigger culling. In particular, the algorithm

makes no assessment of occluder distribution over an occludee and a light weight statistical model may be applicable. Simpler rule bases and metrics may also prove viable, for example, classifying visibility based on occludee overlap with tagged surface edge silhouette occluders or just surface interior occluders to distinguish between *partially visible* and *occluded* states.

Currently, scene nodes at varying resolutions in the hierarchy undergoing refinement, are tested for occlusion by the occlusion mask, which remains at a fixed image space resolution. Further work may investigate the feasibility of a hierarchical occlusion mask representation. Potential may also exist for image graph relation culling, to limit processing to the most important regions of the graph.

The potentially visible set identified by stage two, can support caching schemes. An early GPU based caching extension has been developed, where a cache is constructed for each occlusion mask image node, storing the image nodes resulting from further refinement in stage three. Caches are then directly rendered on the GPU, calculating view cones and splats. These caches are then reused if still valid in subsequent frames, even if changes occur in the occlusion mask. Cache validity is based on screen space tolerance metrics that govern safety zones for the viewpoint. Caching reduces refinement in stage three and also reduces bandwidth between the CPU and GPU. Early results show at least a 4x speedup (see Section 5.8.4). Other approaches such as the Sequential Point Trees system by Dachsbacher and colleagues [48] may also be adapted for use. GPU architecture speeds are set to increase with unified vertex and fragment processors such as in the ATI R500 GPU, potentially improving vertex processing performance that will suit point based rendering systems that use GPUs, including caching schemes.

Further work on procedural scene generation in the framework is suitable, from scales such as cities, to microscopic 3D textures. To support a wide range of scene scales, additional techniques may be required such as normal cone attenuation from microscopic to macroscopic scales and for practical issues such as binary word length limitations.

The system may also be extended to other shading techniques such as ray tracing, using the algorithm to establish surface visibility and exploiting the scene tree hierarchy for faster ray intersection calculations.

7.3 Conclusion

This thesis has presented a simple, extensible point based rendering algorithm with a worst case complexity of $O(j^2)$ in the number of scene leaf primitives j, but typically achieves sub-linear (better than linear) complexity by combining a number of scalability solutions, particularly level of detail and occlusion culling, based on a viewpoint dependent selective refinement system. Although the system scales well, further work is required to speed up frame rates for practical use.

Although the algorithm is approximate, in many test cases it can render images with few visible artifacts, but some scenes still induce noticeable errors, particularly through complex vistas, where objects are visible through a number of closer silhouettes. Future work is required to reduce these errors.

In tests the algorithm has been shown to render scenes consisting of several hundred thousand objects that would conventionally consist of several hundred billion polygons at about a frame per second on a 2.8GHz system with minimal graphics card support with performance considered approximately comparable to other integrated rendering solutions.

Glossary

TABLE 18. Glossary: Terms with * are specific to the algorithm

Term	Description
Advanced Graphics Port	Advanced Graphics Port bus designed for higher speed data transfer between the CPU and GPU.
AGP	(See Advanced Graphics Port.)
Anti-Aliasing	The process of eradicating alternate sampling representations, often formed by the discrete sampling of a continuous signal.
Aspect Graph	Graph of cells which views of an environment which are invariant with respect to which primitives occlude which others.
Binary Space Partition (BSP) Tree	Hierarchical spatial partitioning technique in 2D or 3D that subdivides space into two convex regions recursively.
BSP Tree	(See Binary Space Partition Tree)
Cache	A dataset stored for reuse, usually for a limited period of time.
Cache (L1, L2, L3)	CPU caches used to store frequently used data in memory that is faster for the CPU to access.
Cg	Basic C like programming language developed by nVidia corporation for the development of Vertex and Fragment programs in GPU programming.
Conic Section	Family of shapes formed by the intersection of a cone and a plane. Possible shapes are circle, ellipse, parabola and hyperbola.
Control Node *	Node in the Scene Graph hierarchy which governs some aspect of the rendering or representation of a scene graph, but which does not in itself represent a surface. Examples include Generator and Transform nodes.
CPU	Central Processing Unit
Decimator (Mesh)	Algorithm which removes geometry from models to simplify their representation.

TABLE 18. Glossary: Terms with * are specific to the algorithm

Term	Description
Diffuse	Reflection of light where incident light is scattered equally in all directions over the hemisphere of the point of incidence.
Edge Collapse	Contraction of a mesh edge to merge it's two vertices to a single vertex. It is the inverse of a *vertex split* operation. Used various LOD methods.
Elliptical Weighted Average Filter (EWA)	Filter that reconstructs object space surfaces, mapping into image space with pre-filtering for anti-aliasing.
EWA Filter	(See Elliptical Weighted Average Filter.)
Fragment Program	A GPU program that processes a small geometric surface's contribution to a pixel under projection to screen space.
Frame Period	Period of time taken to rendering a single image, typically measured in milliseconds.
Frame Rate	Rate at which frames of animation are produced. Typical metrics are the frequency (Hz) or the frame period.
Frame Rate Consistency	Amount of deviation from a mean frame rate across a range of periods.
GPU	(See Graphics Processing Unit)
Graphics Processing Unit (GPU)	Advanced processor chip used in modern graphics cards for parallel processing of vertex and fragment programs using an optimized memory bandwidth architecture.
Hysteresis	The level of smooth transitions between models at multiple levels of detail.
Image Based Rendering	A class of rendering methods which permit the rendering of a novel image from one or more existing images.
Image Caching	A class of rendering algorithm which uses altered forms of previously rendered image components to create a new image.
Image Graph *	Graph used to represent occlusion properties between projected scene nodes in the current solution.
Image Node *	Node in the Image Graph representing the image space representation of a Scene Node.
Image Relation *	Arc in the Image Graph representing an image space and object space relationship between two image nodes.
Image Space	Scalar space in which a 2D image is described in the view plane.
Image Tree *	Tree of Image Nodes that describe the refinement of a region of a scene tree when rendering an image from a viewpoint or light source.
Impostors	A method similar to billboards which use textured polygons to replace an area of a 3D environment which will not be rendered for increase performance.

TABLE 18. Glossary: Terms with * are specific to the algorithm

Term	Description
Instance	Application of a sub-scene graph within another scene graph to replicate the same object or scene region one or more times.
K-D Tree	A multi-dimensional search tree in K dimensions. Levels of tree are partitioned on sequential dimensions. (Constrained form of Binary Space Partition Tree)
Least Squares	Linear regression tool that approximates a dataset with a linear form that has minimal distance to all points in the dataset. Common forms are 2D lines and 3D planes.
Level of Detail (LOD) Control	The process of providing single object models in discrete or continuously changing scales of feature, usually bounded by the original model and the lowest topologically correct representation.
Locale Management	System that identifies one or more subsets of the scene to serve as input to the main rendering algorithm to reduce the size of the data set addressed.
Locale Scene Node *	Scene node chosen by the locale manager that serves as a root node in the scene tree from which rendering will begin. The child scene tree of the locale scene node is the only region of the scene tree to be addressed during rendering.
LOD	(See Level of Detail Control)
MLS	(See Moving Least Squares)
Moving Least Squares	Least square technique used for interpolation, smoothing and derivative analysis on scattered data sets.
Normal Cone	A cone that describes a solid angle within which normal vectors are present. Can also bound vector origins.
Normal Vector	Vector (Usually normalized) that is perpendicular to a surface.
Nyquist Frequency (Limit)	Highest frequency that can be present in a signal that will be sampled at a a frequency greater than twice this highest frequency.
Object Space	Spatial co-ordinate system in which an object is defined.
Occludee	An object or surface region that is hidden by another (Occluder) under projection to image space.
Occluder	An object or surface region that hides another (Occludee) under projection to image space.
Occluder Strength *	Estimate of the effectiveness of an occluding image node to occlude other nodes further away from the viewpoint.
Occlusion Culling	The process of removing objects or surface regions in a scene which are not visible to improve rendering times and system scalability.
Occlusion Mask *	Geometric data structure representing coverage of the image by visible surfaces in the scene. This data structure is used to calculate occluded regions of the scene.

TABLE 18. Glossary: Terms with * are specific to the algorithm

Term	Description
Octree	Hierarchical spatial partitioning technique in 3D that sub-divides a box into eight sub-boxes (octants) recursively. (Constrained form of K-D or BSP tree).
P.B.R.	(See Point Based Rendering)
Parallel Visible Area Function (PVA) *	Function that estimates the amount of surface area component to the viewpoint under a parallel projection used to estimate occlusion strength.
Parametric surface	Surface description method using single or multivariate functions which generate surfaces explicitly using a sampling.
PCA	(See Principle Components Analysis.)
Pipeline	Hardware architecture consisting of multiple modules in a chain that pass results sequentially, whilst operating in parallel.
Pitch	Orientation of the virtual camera with respect to viewing up or down on the X axis
Point	Theoretically infinitesimally small position primitive, but often assigned a size in point based rendering.
Point Based Rendering	A rendering method which uses points as a primitive, typically hierarchical in nature.
Popping	The level of noticeable change when models are switched between multiple levels of detail.
Pre-filter	A screen space anti-aliasing technique to band limit frequencies for discrete sampling by the pixel grid.
Principle Components Analysis	Cluster analysis technique to capture the variance in a dataset in terms of principle components, used to reduce the dimensionality of data and summarize the most defining parts, whilst filtering the dataset.
PVA *	(See Parallel Visible Area)
Quadtree	Hierarchical spatial partitioning technique in 2D that sub-divides a box into four sub-boxes (quadrants) recursively.
Radiosity	A rendering method (usually pre-process) which attempts to simulate diffuse inter-reflection in scenes. Substantial levels of visual realism can be obtained. Processing costs are also substantial and though the method is viewpoint independent, the systems must be resolved for scene dynamics. The method is sometimes combined with view dependent specular calculations to realise very realistic images.
Ray Tracing	Rendering technique which traces rays of light as they reflect and refract around a 3D environment. Visually realistic images can be generated, though the method is CPU intensive. Good results can be obtained using scenes with high specularity.
Roll	Orientation of the virtual camera about the Z axis

TABLE 18. Glossary: Terms with * are specific to the algorithm

Term	Description
Scene Graph	A directed graph structure of heterogeneous nodes which perform a variety of functions ranging from high level operations to low level surface representation data. Scene graphs have become a common data structure in rendering systems. Example systems include: VRML, VRML97, Open Inventor, SGI Performer, Java 3D
Scene Node *	A node representing a spherical primitive in a Scene Tree.
Scene Tree *	Hierarchical tree of surface elements which represent a scene at any scale.
Selective Refinement	A level of detail control method by which resolutions across an object's surfaces are chosen at runtime based on the viewpoint, potentially with less pre-processing.
Shader	Function or program module used to calculate light interaction at surfaces.
Solid Angle Occlusion Ratio *	Ratio of coverage of one View Cone solid angle by another, with both cones sharing the same apex. Used to establish the level of occlusion of an Occludee by an Occluder.
Specular	Mirror type reflection of light where the angle of incidence with a surface is equal to the angle of reflection (in perfect reflection).
Splat	A screen space area based primitive that represents the footprint of an object space primitive, often more approximately for speed.
Spline	Surface description method using single or multivariate, non linear functions. Splines are typically lines or surface patches.
Surfel	'Surface Element', a geometric primitive used to represent a surface sample, usually formed in a 2D tangent plane.
Vertex Program	GPU program applied to vertices during the 3D pipeline.
Vertex Split	Expansion of a vertex to an edge, forming new polygons in LOD methods. It is the inverse of an *Edge Collapse* operation.
View Cone	Cone of visibility of an object, with apex at the camera's centre of projection and sides tangential to a bounding sphere around the object.
View Volume	Volume formed by the planes connecting the virtual camera's COP to it's boundaries. Generally this is a four sided pyramid.
Viewpoint	Point from which a scene is viewed by an observer. This is usually intended to be the virtual camera's centre of projection (COP).
Viewspace	The space in which the virtual camera's viewpoint can move. This space may be 3D or 2D and may be constrained to specific regions.
Virtual Root Node	A conceptual root node that may or may not be stored explicitly in a generator. This node is used by parent nodes in hierarchical partitioning tasks.

TABLE 18. Glossary: Terms with * are specific to the algorithm

Term	Description
Volume Rendering	Representation and rendering of scene or object data as a 3D contiguous medium.
Voronoi	Graph formed by the cells of space nearest to points defined in K dimensional space. (Also the dual of the Delaunay triangulation).
Voxel	Small volumetric element used to describe occupancy in space as a scene representation primitive.
Voxelization	The process of converting an alternative scene representation into voxels in a voxel space. Usually applied to polygonal objects.
Voxels	Data structure (3D Grid of box cells) used to represent a solid in volume rendering.
Yaw	Orientation of the virtual camera with respect to direction on Y axis
Z-Buffer	2D Array of values used to perform depth comparisons for hidden surface removal in rendering. The z-buffer has now become a standard feature in all 3D hardware.

A Appendix

Section A.1 overviews the ***software's architecture*** including layer diagrams and classes.

Section A.2 discusses a number of ***implementation issues*** in C++.

Section A.3 specifically examines ***memory management*** requirements for performance.

Section A.4 details the Scene Tree (SCT) ***binary file format*** used for caching generator node scene graphs and read on demand services.

Section A.5 details ***object translation figures*** to supplement figures in Chapter 6.

A.1 System Architecture

The system has been designed to be highly modular, portable and easily extendable. System specific requirements have been restricted to several small components. This section will look at the basic structure of the system in general and several class hierarchies of note.

A.1.1 System Layers

The system's layers are shown in Figure 97. Each of these layers will be now be summarized.

(a) Client application: system and application dependent and communicates with the whole system through the *render interface* layer (b).

(b) Render interface: supports all basic functions for specifying parameters for system configuration, navigation and rendering.

(c) Render manager: performs main rendering functions for one or more images and light sources with shadows. Multiple **image managers** are stored for each image to be rendered.

(d) Navigation manager: Manages multiple registered navigation systems that are selectable by the application and used in various dynamic processes such as collision detection. The main navigation system in use is based on a walking metaphor.

(e) Image manager: stores all information about an image during rendering, whether a viewpoint or light source image. An **image graph** and an **image tree** are stored, with all rendering parameters required by the render manager.

(f) Draw manager: performs drawing operations in using methods independent of the underlying standard graphics API.

(g) Draw renderer: performs drawing operations using methods dependent on the underlying standard graphics API, with a standardized interface to the draw manager above.

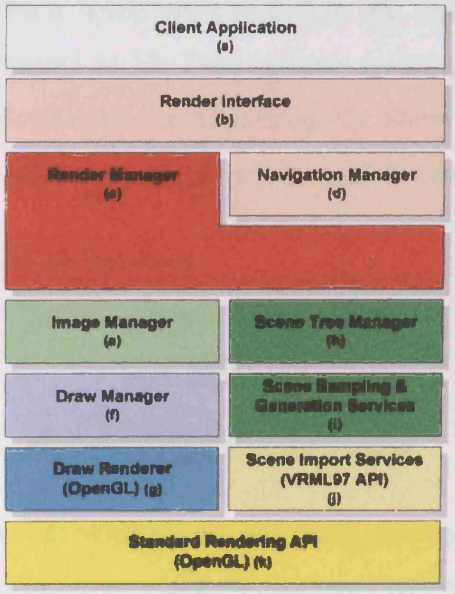
(h) Scene tree manager: stores entire scene graph that on aggregate, describes a scene tree describing the working set. All scene graph nodes including generators are within this layer.

(i) Scene sampling & generation services: provides services to generators for scene sampling using mesh vertex or voxelization methods and hierarchical scene tree construction with and without container packing optimizations.

(j) Scene import services: responsible for importing external scene representation formats. This module currently contains a custom VRML97 scene graph rendering engine.

(k) Standard rendering API: rendering using standard 3D polygon based API. Currently OpenGL is supported, but future versions may also support Microsoft Direct3D or potentially a more basic splat rasterization API.

FIGURE 97. System layer diagram



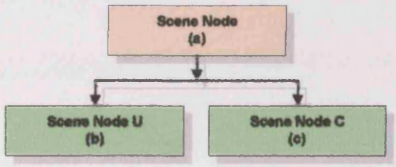
A.1.2 Scene Graph Class Hierarchy

One of the important class hierarchies in the design is that of the scene graph and scene nodes. The scene node hierarchy is simple, as shown in Figure 98. An abstract base class *scene_node* (a) is the parent of two forms in use, currently an un-compressed node (b) and a compressed node (c). In the implementation, the un-compressed form inherits from

multiple classes that support specific attributes to enable future nodes to select those that they require.

The scene graph class hierarchy is shown in Figure 99. A choice was taken to use C++ virtual functions, incurring a small overhead at runtime, to gain substantial functionality and structure in the design. A side effect of this choice is the inclusion of class IDs in each instantiated object, implemented in most systems using a virtual function table pointer. A base class *scene_node_control* (a) is at the root, with *scene_node_container* (b) as the base class for all scene graph nodes that literally or conceptually store at least one scene node. A number of other C++ template container classes are derived from (b). The first is the *scene_node_container_single_h0* that stores a single scene node with two child nodes. This container has a height of zero. This is a special case intended for regions of the scene graph that are quite dynamic, where container packing will not be carried out. Most links between containers in the scene graph refer to two nodes for memory efficiency, but this container points to two. A series of container classes are automatically created for each scene node added to the system using templates, with heights currently ranging from 1 to 12. Two variants are present for all 12, one for leaf containers (e) and one derived class for branching (f). There are therefore currently 50 different container classes instantiated in the system, 25 for each scene node type.

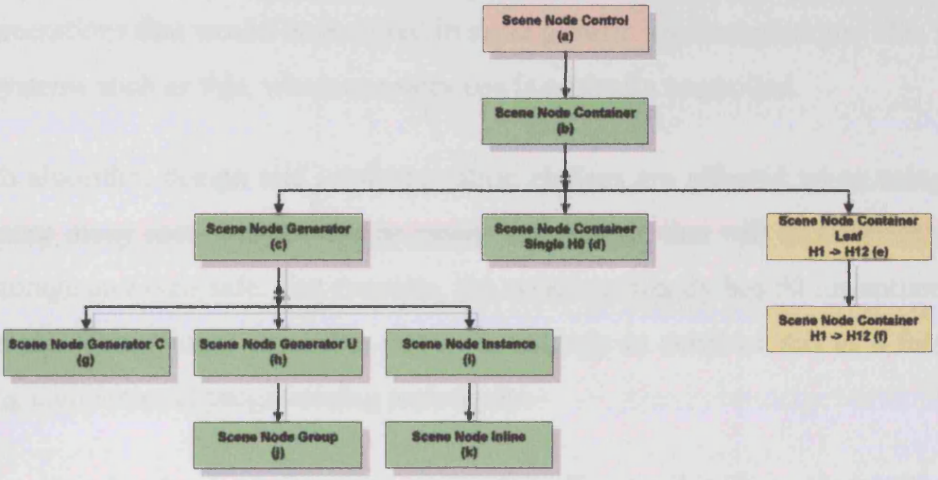
FIGURE 98. Basic scene node class hierarchy



A base class for generators is provided by *scene_node_generator* (c). This provides general services for local scene graph memory management and read on demand from backing store. Generators for specific node types are then derived in (g) and (h) for compressed and un-compressed nodes respectively. Group nodes are expected to perform on the fly hierarchical partitioning and therefore this class currently derives from the un-compressed form to reduce runtime overheads, but could be derived from the compressed form.

Instancing nodes are considered to be simple forms of procedural generator and are therefore derived from the generator base class (c). These nodes apply filtered transformations on their scene graph to position an instance of their child scene graph which may or may not be shared with other instance nodes. Inline nodes also provide a similar functionality, but also fetch a child generator object from a specified file, so the inline node class is derived from the instance class.

FIGURE 99. Basic scene graph node class hierarchy



The actual implementation splits some of these classes into other classes and are formed through multiple inheritance, but this synopsis is accurate. This splitting of functionality provides greater extendability in future, allowing new node classes to be created from multiple smaller classes.

A.2 Implementation Issues in C++

A.2.1 C++ Template Programming

Template class programming has played an important part, not just in the implementation of the system, but also in the design choices made when developing the algorithm. Templates offer a simple form of Turing complete meta programming that offers substantial implementation and memory efficiencies. A primary use is to create a range of classes with specific sets of data members, without the need for pointers or object type enumerations that would be required in more generic implementations. This is important in systems such as this, where memory use is critically controlled.

Both algorithm design and implementation choices are affected when using templates, because many more classes can be easily instantiated, that will be efficient with respect to storage and type safe. For example, the system currently has 50 instantiated container classes. Conventional algorithm design is unlikely to consider this as a feasible option using conventional programming techniques.

The system has been developed in a way that offers toolkit like extendability using templates. One example is the addition of a new scene node class, where a number of effects occur:

- Internal and branching container class hierarchies are created for the node
- Node is incorporated into the SCT file format
- Basic generator class can be created for specific use of the node, with scene tree construction
- Memory management added for the node type within the generator

Limitations in some compilers, including Microsoft Visual C++ .Net require template class methods to be inline, defined in the header files. This can greatly increase rebuild times, but as a side effect, may result in slightly faster execution.

A.3 Memory Management

Memory management issues are critical in realtime systems. Constant calls for heap allocation can be prohibitively expensive for all objects required in a system. The frequency of calls is generally the most important issue. This section will briefly discuss some simple approaches to managing memory.

A.3.1 Fast Allocation & De-allocation

All data types in the system have been developed with support for custom memory allocation systems. In fact, the memory allocation classes can themselves, have memory allocation classes. However, they typically rely on a default heap allocation class to provide a single level of allocation strategy.

The rendering system requires fast storage allocation and de-allocation in a number of places in particular, including a generator's scene graph nodes and image graph components.

Generators are responsible for creating their own scene graphs and likewise should be responsible for deleting them when required. Memory management systems are specifically customized for each generator type, based on the nodes they intend allocating. These classes can be registered when the generator is created at runtime.

Image nodes in the image tree and image graph may be persistent between frames for coherence based speedups or may be deleted when no longer required. Image relations are currently not maintained between frames and are all de-allocated between frames.

Fast allocation could be achieved in various ways. The current implementation frequently uses block allocation of sequences of arrays that are transparently mapped to a contiguous array. Occasionally, stretch array allocation is used where the overheads of copying array data to larger allocations is less important than the access speed provided by contiguous logical memory.

Memory is often pre-allocated, or allocated on first demand, based on the expected number of instances required of a particular class.

Fast de-allocation is extremely important. For example, currently, all image graph relations are de-allocated between frames ready for the next frame. This can simply be performed in a single function to the memory management systems which is far more efficient than explicitly de-allocating large numbers of objects. This memory is only conceptually de-allocated, but retained for reuse.

A.3.2 Image Tree Management

Image trees nodes may be persistent across frames. Allocation is required to add new nodes to an image's image tree and de-allocation is required when some branches of the image tree are no longer required.

Image trees may reside in memory for as long as is practicable. They may be persistent even for long periods of being off screen, for example when the viewpoint looks away from a surface, or when detail that has previously been viewed is not refined to and is no longer contributing to the image. The system incorporates image tree pruning policies based on *least recently used* de-allocation. These systems are still undergoing development and will not be discussed in detail.

A.4 The Scene Tree (SCT) Binary File Format

The Scene Tree (SCT) binary file format stores a scene graph describing a scene tree in an efficient binary format. Services are provided to allow generators to be capable of reading a specified number of scene tree node levels on demand.

A.4.1 Overview

The SCT binary file format provides a backing store caching system for generators. Files are expected to be hierarchically consistent, i.e. their root node accurately reflects the hierarchical attributes of size, position, surface area and colour etc. that are required by client scene graphs.

The scene graph layout on disk can logically be very similar to that in memory. In fact, they can be so similar that pointers to child nodes from parent nodes in memory and on disk are actually mixed. Child pointers within a scene graph, normally point to scene graph nodes in memory. However, scene graph nodes at the bottom of the working set are flagged as *last resident*. These have file pointers to positions on disk if they are not leaves.

An assumption is made that nodes will be accessed level by level, rather than depth first. The same assumption is also made by QSplat's file format [171] [172]. However, this will only be true to an extent. Other considerations in this system, such as back face culling will omit portions of objects, as will occlusion culling that may result in more depth first access patterns.

To facilitate a predominantly breadth first access with support for depth first access, any node on disk can be accessed when required, based on access through memory pointers. This form of access is quite fast while file regions accessed are in disk caches, but will be slower when fetched from disk hardware. Substantial benefits are gained however, because the access patterns can be arbitrary, traversed from disk as required.

Rather than consider reading and writing functions of the file to be based on scene graph structure, functions are instead based on a accessing the scene tree embedded and described by the scene graph in the file. Whilst scene nodes are stored level by level

down the scene tree, some scene graph structure is written in a depth first ordering so that scene node containers required in the breadth first ordering can be accessed.

For cross platform support, mappings between system hardware specific data storage formats are performed automatically using system specific stream modules derived from standard C++ file stream classes. For example, integer big/small endian issues are dealt with.

A.4.2 File Structure

The file begins with header information stating a standard ID string and version number. The first scene graph node in all SCT files is the generator that was responsible for creating the file. This can be used to reconstruct the generator with all its parameters if required. The generator's scene graph then follows.

A.4.3 Scene Graph Node Storage

Scene graph control nodes are stored using the data fields shown in Table 19. The first

TABLE 19. Scene graph control node format

Scene_Node_Control Derived Class Data
Scene_Node_Control class ID
Parent node pointer
Child file Pointers <1...*n*>
Node data (Custom)

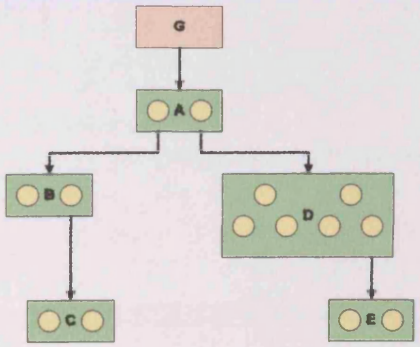
entry is a class ID. From this, when read, the correct object can be allocated ready for data to be read and set. Each class type will have differing numbers of children. For example, the container nodes for heights 1 through to 12 have two to 4096 children. Some nodes, such as leaf containers will have no children at all. The parent pointer then provides a link for traversal up the scene graph. Child pointers are included first to allow algorithms to skip node data if required, to access the children without parsing the node's contents. Traversal without parsing is only possible when delta encoding for compression is not used in scene nodes. Finally, custom node data is written to the file. This is parsed by the node itself on reading, when created by the file importer based on the stated ID.

A.4.4 Read on Demand Services for Out of Core Rendering

A working set scene graph is resident in memory. If any of its terminal nodes are not in memory and are available on disk, they are flagged as being the last resident nodes. Their pointers are then considered file pointers within a file specified by the current traversal status information created and managed during traversal. The traversal status is modified to refer to the file by the generator that owns the nodes and has chosen to use read on demand services.

Given a single starting node, a specified number of scene tree levels can be read from the terminals of the given scene tree. This given node may or may not be the last resident node in memory. If it is, the additional levels are immediately read as its children. If the specified node already has a sub-tree of its own, these scene tree levels are traversed (through the scene graph representation) until the last resident nodes are found. The specified number of levels is then read from these nodes.

FIGURE 100. Scene tree and scene graph depth differences



Scene graphs on disk are accessed based on scene tree levels. Consider the simple example shown in Figure 100 that demonstrates the basic difference between scene tree and scene graph levels. The generator node G has child container A. The container A has two child containers B and D with heights 1 and 2 respectively. Further child containers C and E exist further down. The process of reading a specified number of scene tree levels from this scene graph is clearly different from that of reading a number of scene graph levels. For example, to read 3 scene tree levels, nodes A, B, C and D must be accessed, but not E as its nodes belong to tree level 4.

If any other scene graph control nodes are present between the containers, they are read in a depth first manner until the required level of nodes has been accessed for the scene tree.

The number of levels read on each update is dependent on what the generator requests. Currently there is no concept of speculative pre-fetching other than the reading of multiple levels until the next time data is required and no de-allocation policy for reducing the working set in memory.

A.5 Object Translation Details

A.5.1 Bunny (Textured) [Stanford Computer Graphics Laboratory]

TABLE 20. Bunny - Translation details

Feature	Value
Density	$k = 3, \gamma = 2$
Original Vertices	34,834
Original Triangles	69,451
Voxelized Scene Nodes	2,221,573
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	1,401,494
Scene Nodes per Polygon (Mean)	20.18
Scene Tree Nodes (Total)	2,802,986
SCT File Size	37.23 MB
Number of Containers	636,275
Scene Graph Child Pointers (32 bit)	1,033,124

TABLE 21. Bunny - Translation performance

Translation Aspect	Time (Seconds)
Read (VRML97)	7.147
Voxelize	2.184
Hierarchy Stage 1	12.997
Hierarchy Stage 2	3.256
Total Hierarchy	16.253
Write (SCT binary file format)	23.474
Total (Voxelization & Hierarchy)	18.437
Total (Voxelization & Hierarchy, Read & Write)	49.058

TABLE 22. Bunny - Scene graph containers (branching and leaf)

Scene Graph Container Type	No. Containers
H = 1, Branching	100,824
H = 2, Branching	90,675
H = 3, Branching	16,285
H = 4, Branching	4,060
H = 5, Branching	624
H = 6, Branching	218
H = 7, Branching	206
H = 8, Branching	355
H = 9, Branching	159
H = 10, Branching	38
H = 11, Branching	1
H = 12, Branching	0
H = 1, Leaf	352,916
H = 2, Leaf	65,819
H = 3, Leaf	3,764
H = 4, Leaf	323
H = 5, Leaf	8
H = 6, Leaf	0
H = 7, Leaf	0
H = 8, Leaf	0
H = 9, Leaf	0
H = 10, Leaf	0
H = 11, Leaf	0
H = 12, Leaf	0

A.5.2 Igea Venus [Cyberware Inc.]

TABLE 23. Venus - Translation details

Feature	Value
Density	$k = 2, \gamma = 2$
Original Vertices	134,345
Original Triangles	268,686
Voxelized Scene Nodes	3,004,914
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	1,339,598
Scene Nodes per Polygon (Mean)	4.99
Scene Tree Nodes (Total)	2,679,194
SCT File Size	35.59 MB
Number of Containers	613,101
Scene Graph Child Pointers (32 bit)	984,676

TABLE 24. Venus - Translation performance

Translation Aspect	Time (Seconds)
Read (VRML97)	7.506
Voxelize	3.708
Hierarchy Stage 1	16.248
Hierarchy Stage 2	3.674
Total Hierarchy	19.922
Write (SCT binary file format)	24.946
Total (Voxelization & Hierarchy)	23.630
Total (Voxelization & Hierarchy, Read & Write)	56.082

TABLE 25. Venns - Scene graph containers (branching and leaf)

Scene Graph Container Type	No. Containers
H = 1, Branching	113594
H = 2, Branching	80426
H = 3, Branching	7367
H = 4, Branching	1015
H = 5, Branching	575
H = 6, Branching	1435
H = 7, Branching	1584
H = 8, Branching	166
H = 9, Branching	2
H = 10, Branching	0
H = 11, Branching	0
H = 12, Branching	1
H = 1, Leaf	331705
H = 2, Leaf	74357
H = 3, Leaf	852
H = 4, Leaf	21
H = 5, Leaf	1
H = 6, Leaf	0
H = 7, Leaf	0
H = 8, Leaf	0
H = 9, Leaf	0
H = 10, Leaf	0
H = 11, Leaf	0
H = 12, Leaf	0

A.5.3 Isis [Cyberware Inc.]

TABLE 26. Isis - Translation details

Feature	Value
Density	$k = 2, \gamma = 2$
Original Vertices	187,644
Original Triangles	375,284
Voxelized Scene Nodes	5,344,016
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	2,614,887
Scene Nodes per Polygon (Mean)	6.97
Scene Tree Nodes (Total)	5,229,772
SCT File Size	67.95 MB
Number of Containers	1,083,271
Scene Graph Child Pointers (32 bit)	1,581,856

TABLE 27. Isis - Translation performance

Translation Aspect	Time (Seconds)
Read (VRML97)	10.075
Voxelize	5.820
Hierarchy Stage 1	29.255
Hierarchy Stage 2	6.713
Total Hierarchy	35.968
Write (SCT binary file format)	42.484
Total (Voxelization & Hierarchy)	41.788
Total (Voxelization & Hierarchy, Read & Write)	94.347

TABLE 28. Isis - Scene graph containers (branching and leaf)

Scene Graph Container Type	No. Containers
H = 1, Branching	127,672
H = 2, Branching	67,488
H = 3, Branching	8,576
H = 4, Branching	1,413
H = 5, Branching	241
H = 6, Branching	331
H = 7, Branching	1,140
H = 8, Branching	1,946
H = 9, Branching	523
H = 10, Branching	20
H = 11, Branching	0
H = 12, Branching	1
H = 1, Leaf	700,192
H = 2, Leaf	169,552
H = 3, Leaf	3,852
H = 4, Leaf	313
H = 5, Leaf	11
H = 6, Leaf	0
H = 7, Leaf	0
H = 8, Leaf	0
H = 9, Leaf	0
H = 10, Leaf	0
H = 11, Leaf	0
H = 12, Leaf	0

A.5.4 Female [Cyberware Inc.]

TABLE 29. Female - Translation details

Feature	Value
Density	$k = 2, \gamma = 1.5$
Original Vertices	302,948
Original Triangles	605,086
Voxelized Scene Nodes	7,850,883
Voxelized Scene Nodes (Filtered) / Scene Tree Leaf Nodes	3,720,501
Scene Nodes per Polygon (Mean)	6.15
Scene Tree Nodes (Total)	7,441,000
SCT File Size	100 MB
Number of Containers	1,708,822
Scene Graph Child Pointers (32 bit)	2,766,962

TABLE 30. Female - Translation performance

Translation Aspect	
Read (VRML97)	18.003
Voxelize	7.476
Hierarchy Stage 1	44.300
Hierarchy Stage 2	9.055
Total Hierarchy	53.355
Write (SCT binary file format)	64.745
Total (Voxelization & Hierarchy)	60.831
Total (Voxelization & Hierarchy, Read & Write)	143.579

TABLE 31. Female - Scene graph containers (branching and leaf)

Scene Graph Container Type	No. Containers
H = 1, Branching	289273
H = 2, Branching	245086
H = 3, Branching	37,683
H = 4, Branching	9,343
H = 5, Branching	1,726
H = 6, Branching	809
H = 7, Branching	1,237
H = 8, Branching	1,273
H = 9, Branching	296
H = 10, Branching	10
H = 11, Branching	0
H = 12, Branching	1
H = 1, Leaf	925,002
H = 2, Leaf	191,791
H = 3, Leaf	4,953
H = 4, Leaf	330
H = 5, Leaf	9
H = 6, Leaf	0
H = 7, Leaf	0
H = 8, Leaf	0
H = 9, Leaf	0
H = 10, Leaf	0
H = 11, Leaf	0
H = 12, Leaf	0

References

[1] B. Adams, P. Dutré.
Interactive Boolean Operations On Surfel-bounded Solids.
ACM Transactions on Graphics 22(3) 2003, pp. 651-656.

[2] B. Adams, R. Keiser, M. Pauly, L. Guibas, M. Gross, P. Dutré.
Efficient Raytracing of Deforming Point-Sampled Surfaces.
Computer Graphics Forum 24(3) 2005, pp. 677-684.

[3] A. Adamson, M. Alexa.
Approximating Bounded Non-orientable Surfaces from Points.
Shape Modeling International proceedings 2004, pp. 243-252.

[4] A. Adamson, M. Alexa.
Ray Tracing Point Set Surfaces.
IEEE International Conference on Shape Modeling and Applications 2003, pp. 272-279.

[5] E. H. Adelson, J.R. Bergen.
The Plenoptic Function and the Elements of Early Vision.
Computation Models of Visual Processing, published by MIT Press, Cambridge, 1991.

[6] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, C. T. Silva.
Point Set Surfaces.
IEEE Visualization proceedings 2001, pp. 21-28.

[7] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, D. Manocha.
MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration.
ACM Symposium on Interactive 3D Graphics 1999, pp. 199-206.

[8] N. Amenta, Y. J. Kil.
Defining Point-Set Surfaces.
Computer Graphics, SIGGRAPH proceedings 2004, pp. 264-270.

[9] C. Andújar, C. Saona-Vázquez, I. Navazo, P. Brunet.
Integrating Occlusion Culling and Levels of Detail through Hardly-Visible Sets.
Computer Graphics Forum 19(3) 2000, pp. 499-506.

[10] W. Aref, H. Samet.
An Algorithm For Perspective Viewing Of Objects Represented By Octrees.
Computer Graphics Forum 14(1) 1995, pp. 59-66.

[11] K. Bala, B. Walter, D. P. Greenberg.
Combining Edges and Points for Interactive High-Quality Rendering.
ACM SIGGRAPH proceedings 2003, pp. 631-640.

[12] R. Baptista.
Compressed Quantized Unit Vectors.
GameDevNet online article.

[13] J. Barrus, R. Waters, D. Anderson.
Locales: Supporting Large Multiuser Virtual Environments.
IEEE Computer Graphics and Applications 16(6) 1996, pp. 50-57.

[14] W. Baxter, A. Sud, N. Govindaraju, D. Manocha.
Gigawalk: Interactive Walkthrough Of Complex Environments.
Eurographics 13'th Workshop on Rendering proceedings 2002, pp. 203-214.

[15] J. Bentley
Multidimensional Search Trees Used For Associative Searching.
Communications of the ACM 18(9) 1975, pp. 509-517.

[16] J. Bittner, V. Havran.
Exploiting Temporal and Spatial Coherence In Hierarchical Visibility Algorithms.
Spring Conference on Computer Graphics (SCCG) 2001, pp. 213-220.

[17] J. Bittner, V. Havran, P. Slavik.
Hierarchical Visibility Culling With Occlusion Trees.
IEEE Computer Graphics International proceedings 1998, pp. 207–219.

[18] J. Blinn.
Texture and Reflection in Computer Generated Images.
Communications of the ACM 19(10) 1976, pp. 542-547.

[19] M. R. Bolin, G. W. Meyer.
A Perceptually Based Adaptive Sampling Algorithm.
Computer Graphics, SIGGRAPH proceedings 1998, pp. 299-310.

[20] M. Botsch, A. Hornung, M. Zwicker, L. Kobbelt.
High-Quality Surface Splatting on Today's GPUs.
IEEE Eurographics Symposium on Point-Based Graphics 2005, pp. 17-24.

[21] M. Botsch, M. Spernat, L. Kobbelt.
Phong Splatting.
IEEE Eurographics Symposium on Point Based Graphics 2004, pp. 25-34.

[22] M. Botsch, A. Wiratanaya, L. Kobbelt.
Efficient High Quality Rendering of Point Sampled Geometry.
Eurographics 13'th Workshop on Rendering proceedings 2002, pp. 53-64.

[23] G. Bradshaw, C. O'Sullivan.
Sphere-tree Construction Using Dynamic Medial Axis Approximation.
ACM Symposium on Computer Animation 2002, pp. 33-40.

[24] L. Bull. M. Slater.
Canopy: Point Based Rendering with Unified Scalability Solutions.
Submitted 2006.

[25] E. Catmull.
A Subdivision Algorithm for Computer Display of Curved Surfaces.
Thesis 1974, University of Utah.

[26] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, J. Snyder.
Fast Rendering Of Complex Environments Using A Spatial Hierarchy.
Graphics Interface 1996 Canadian Human-Computer Communications Society 1996, pp. 132–141.

[27] S. Chen.
QuickTime VR: An Image-Based Approach to Virtual Environment Navigation.
Computer Graphics, SIGGRAPH proceedings 1995, pp. 29-38.

[28] S. Chen, L. Williams.
View Interpolation for Image Synthesis.
Computer Graphics, SIGGRAPH proceedings 1993, pp. 279-288.

[29] B. Chen, M. Xuan Nguyen.
Pop: A Hybrid Point and Polygon Rendering System For Large Data.
ACM Visualization proceedings 2001, pp. 45-52.

[30] N. Chin, S. Feiner.
Near Real-time Shadow Generation Using BSP Trees.
Computer Graphics, SIGGRAPH proceedings 1989, pp. 99-106.

[31] Y. Cho, J. Woo.
Improved Specular Highlights With Adaptive Shading.
IEEE Computer Graphics International proceedings 1996, pp. 38-46.

[32] Y. Chrysanthou, M. Slater
Shadow Volume BSP Trees For Computation Of Shadows In Dynamic Scenes.
ACM Symposium on Interactive 3D Graphics Proceedings 1995, pp. 45-50.

[33] P. Cignoni, C. Rocchini, R. Scopigno.
Metro: Measuring Error On Simplified Surfaces.
Computer Graphics Forum, 17(2) 1998, pp. 167–174.

[34] U. Clarenz, M. Rumpf, A. Telea.
Fairing of Point Based Surfaces.
Computer Graphics International Proceedings 2004, pp. 600-603.

[35] J. Clark.
Hierarchical Geometric Models for Visible Surface Algorithms.
Communications of the ACM 19(10) 1976, pp. 547-554.

[36] L. Coconu, H. Hege.
Hardware-oriented Point-based Rendering Of Complex Scenes.
Eurographics 13'th Workshop on Rendering Proceedings 2002, pp. 43-52.

[37] J. Cohen, D. G. Aliaga, W. Zhang.
Hybrid Simplification: Combining Multi-resolution Polygon and Point Rendering.
IEEE Visualziation proceedings 2001, pp. 37-44.

[38] J. Cohen, M. Olano, D. Manocha.
Appearance-Preserving Simplification.
Computer Graphics, SIGGRAPH proceedings 1998, pp. 115-122.

[39] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W. Wright.
Simplification Envelopes.
Computer Graphics, SIGGRAPH proceedings 1996, pp. 119–128.

[40] D. Cohen-Or, Y. Chrysanthou, C. T. Silva, F. Durand.
A Survey of Visibility for Walkthrough Applications.
Eurographics proceedings, course notes 2000.

[41] D. Cohen-Or, G. Fibich, D. Halperin, E. Zadicario.
Conservative Visibility and Strong Occlusion For Viewspace Partitioning Of Densely Occluded Scenes.
Computer Graphics Forum, 17(3) 1998, pp. 243-254.

[42] D. Cohen-Steiner, E. Colin de Verdiere, M. Yvinec.
Conforming Delaunay Triangulations in 3D.
ACM Symposium on Computational Geometry proceedings 2002, pp. 199-208.

[43] S. Coorg, S. Teller.
Real-time Occlusion Culling For Models With Large Occluders.
Available: ACM Symposium on Interactive 3D Graphics Proceedings 1997, pp. 83–90.

[44] S. Coorg, S. Teller.
Temporally Coherent Conservative Visibility (extended abstract).
Symposium on Computational Geometry 1996, pp. 78–87.

[45] W. Correa, J. Klosowski, C. Silva.
iWalk: Interactive Out-of-Core Rendering of Large Models.
Technical Report TR-653-02, Princeton University, 2002.

[46] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, M. Howard.
Towards an Interactive High Visual Complexity Animation System.
Computer Graphics 13(2) 1979, pp. 289-298.

[47] F. Dachille 1X, A. Kaufman
Incremental Triangle Voxelization
Graphics Interface 2000, pp. 205-212.

[48] C. Dachsbacher, C. Vogelgsang, M. Stamminger.
Sequential Point Trees.
Computer Graphics, SIGGRAPH proceedings 2003, pp. 657-662.

[49] M. Deering
Geometry Compression.
Computer Graphics, SIGGRAPH proceedings 1995, pp. 13-20.

[50] T. D. DeRose, M. Lounsbery, J. Warren.
Multiresolution Analysis for Surfaces of Arbitrary Topological Type.
ACM Transactions on Graphics, 16(1) 1997, pp. 34-74.

[51] Y. Dobashi, T. Yamamoto, T. Nishita.
Radiosity for Point-Sampled Geometry.
IEEE Pacific Graphics proceedings 2004, pp. 152-159.

[52] F. Duguet, G. Drettakis.
Flexible Point-Based Rendering on Mobile Devices.
IEEE Computer Graphics and Applications 24(4) 2004, pp. 57-63.

[53] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, W. Stuetzle.
Multiresolution Analysis of Arbitrary Meshes.
Computer Graphics, SIGGRAPH proceedings 1995, pp. 173–182.

[54] J. El-Sana, N. Sokolovsky, C. Silva.
Integrating Occlusion Culling With View-dependent Rendering.
IEEE Visualization Proceedings 2001, pp. 371-378.

[55] C. Erikson.
Polygonal simplification: An overview.
Technical Report TR96-016, 16, 1996 Computer Sceience, University of North Carolina, Chapel Hill.

[56] C. Erikson, D. Manocha.
GAPS: General and Automatic Polygonal Simplification.
ACM Symposium on Interactive 3D Graphics 1999, pp. 79-88.

[57] C. Erikson, D. Manocha, W. Baxter.
HLODs for Faster Display of Large Static and Dynamic Environments.
ACM Symposium on Interactive 3D Graphics 2001, pp. 111-120.

[58] C. Everitt, A. Rege, C. Cebenoyan.
Hardware Shadow Mapping.
White paper nVidia corporation.

[59] P. Fearing.
Computer Modelling Of Fallen Snow.
Computer Graphics, SIGGRAPH proceedings 2000, pp. 37-46.

[60] R. Fernando, M. J. Kilgard.
The Cg Tutorial: The Definitive Guide to Programmable Real-time Graphics.
Addison Wesley 2003, ISBN 0321194969.

[61] S. Fleishman.
Point Set Surfaces.
Ph.D. thesis 2003, School of Computer Science, Tel-Aviv University.

[62] S. Fleishman, D. Cohen-Or, M. Alexa, C. T. Silva.
Progressive Point Set Surfaces.
ACM Transactions on Graphics, 22(4), 2003. pp. 997-1011.

[63] J. Foley, A. Van Dam, S. Feiner, J. Hughes.
Computer Graphics, Principles and Practice In C.
Addison Wesley 2003, ISBN: 0321210565

[64] H. Fuchs, Z. M. Kedem, B. F. Naylor.
On Visible Surface Generation by a Priori Tree Structures.
Computer Graphics, SIGGRAPH proceedings 1980, pp. 124-133.

[65] T. Funkhouser, D. Khorramabadi, C. Sequin, S. Teller.
The UCB System for Interactive Visualization of Large Architectural Models.
Presence 5(1) 1996, pp. 13-44.

[66] T. Funkhouser, C. Sequin.
Adaptive Display Algorithm for Interactive Frame Rates During Visualization Of Complex Virtual Environments.
Computer Graphics, SIGGRAPH proceedings 1993, pp. 247–254.

[67] M. Garland.
Multiresolution Modeling: Survey and Future Opportunities.
Eurographics State of the Art Reports 1999, pp. 111-131.

[68] M. Garland, P. Heckbert.
Simplifying Surfaces With Color and Texture Using Quadric Error Metrics.
IEEE Visualization proceedings 1998, pp. 263-270.

[69] M. Garland, P. Heckbert.
Surface Simplification Using Quadric Error Metrics.
Computer Graphics, SIGGRAPH proceedings 1997, pp. 209–216.

[70] A. Glassner (Editor).
An Introduction To Ray Tracing (4th edition).
Academic Press 1991, ISBN 0-12-286160-4.

[71] E. Gobbetti, F. Marton.
Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms.
Computer Graphics, SIGGRAPH proceedings 2005, pp. 878-885.

[72] E. Gobbetti, F. Marton.
Layered Point Clouds.
IEEE Eurographics Symposium on Point Based Graphics 2004, pp. 113-120.

[73] C. Goral, K. Torrance, D. Greenberg, B. Battaile.
Modelling The Interaction of Light Between Diffuse Surfaces.
Available: Computer Graphics, SIGGRAPH proceedings 1984, pp. 212-222.

[74] S. J. Gortler, R. Grzeszczuk, R. Szeliski, M. F. Cohen.
The Lumigraph.
Computer Graphics, SIGGRAPH proceedings 1996, pp. 43-54.

[75] S. Gottschalk, M. C. Lin, D. Manocha.
OBB-Tree: A Hierarchical Structure for Rapid Interference Detection.
Computer Graphics, SIGGRAPH proceedings 1996, pp. 171-180.

[76] N. Govindaraju, A. Sud, S. Yoon, D. Manocha.
Interactive Visibility Culling in Complex Environments using Occlusion-Switches.
ACM Symposium on Interactive 3D Graphics proceedings 2003, pp. 103-112.

[77] N. Greene.
Hierarchical Polygon Tiling With Coverage Masks.
Available: Computer Graphics, SIGGRAPH proceedings 1996, pp. 65–74.

[78] N. Greene, M. Kass, G. Miller.
Hierarchical Z-buffer Visibility.
Computer Graphics, SIGGRAPH proceedings 1993, pp. 231–240.

[79] J. P. Grossman.
Point Sample Rendering.
MSc. Thesis 1996, Electrical Engineering and Computer Science, University of Toronto.

[80] J. P. Grossman, W. Dally.
Point Sample Rendering.
Eurographics 9'th Workshop on Rendering proceedings 1998, pp. 181-192.

[81] G. Guennebaud, L. Barthe, M. Paulin.
Deferred Splatting.
Computer Graphics Forum 23(3) 2004, pp. 653-660.

[82] G. Guennebaud, L. Barthe, M. Paulin.
Interpolatory Refinement for Real-Time Processing of Point-Based Geometry.
Computer Graphics Forum 24(3) 2005, pp. 657-666.

[83] G. Guennebaud, L. Barthe, M. Paulin.
Real-Time Point Cloud Refinement.
IEEE Eurographics Symposium on Point-Based Graphics 2004, pp. 41-48.

[84] G. Guennebaud, M. Paulin.
Efficient Screen Space Approach for Hardware Accelerated Surfel Rendering.
Vision, Modeling and Visualization proceedings 2003, pp. 1-10.

[85] S. Gumhold, X. Wang, R. Macleod.
Feature Extraction from Point Clouds.
10th Int. Meshing Roundtable 2001, pp. 293-305.

[86] X. Guo, J. Hua, H. Qin.
Point Set Surface Editing Techniques Based on Level-Sets.
IEEE Computer Graphics International proceedings 2004, pp. 52-59.

[87] X. Guo, H. Qin.
Dynamic Sculpting and Deformation of Point Set Surfaces.
Pacific Graphics 2003, pp. 123-130.

[88] M. Guthe, P. Borodin, P. Balazs, R. Klein.
Real-time Appearance Preserving Out-of-Core Rendering with Shadows.
Eurographics Symposium on Rendering 2004, pp. 69-79, 409.

[89] S. Guthe, M. Wand, J. Gonser, W. Straßer.
Interactive Rendering of Large Volume Data Sets.
IEEE Visualization proceedings 2002, pp. 53-60.

[90] E. Haines, T. Moller.
Real-time Shadows.
Games Developers Conference proceedings 2001.

[91] B. Hamann.
A Data Reduction Scheme For Triangulated Surfaces.
Computer Aided Geometric Design 11(2) 1994, pp. 197-214.

[92] M. Harris, A. Lastra.
Real-Time Cloud Rendering.
Eurographics proceedings 20(3) 2001, pp. 76-84.

[93] J. Hasenfratz, M. Lapierre, N. Holzschuch, F. Sillion.
A Survey of Real-Time Soft Shadows Algorithms.
Computer Graphics Forum 22(4) 2003, pp. 753-774.

[94] P. Heckbert.
Fundamentals of Texture Mapping and Image Warping.
Masters Thesis 1989, Dept. Electrical Engineering and Computer Science, University of California, Berkeley.

[95] P. Heckbert, M. Garland.
Survey of Polygonal Surface Simplification Algorithms.
Technical report CMU-CS-95-194 1995, Computer Science, Carnegie Mellon University.

[96] H. Hey, R. F. Tobler, W. Purgathofer.
Real-Time Occlusion Culling with a Lazy Occlusion Grid.
Eurographics 12'th Workshop on Rendering proceedings 2001, pp. 215-220.

[97] K. Hillesland, B. Salomon, A. Lastra, D. Manocha.
Fast and Simple Occlusion Culling Using Hardware-based Depth Queries.
Tech Report TR02-039 2002, Department of Computer Science, University of North Carolina.

[98] M. Hopf, M. Luttenberger, T. Ertl.
Hierarchical Splatting of Scattered 4D Data.
IEEE Computer Graphics and Applications 24(4) 2004, pp. 64-72.

[99] H. Hoppe.
Efficient Implementation of Progressive Meshes.
Available: Computers and Graphics, 22(1) 1998, pp. 27–36.

[100] H. Hoppe.
Progressive Meshes.
Available: Computer Graphics, SIGGRAPH proceedings 1996, pp. 99–108.

[101] H. Hoppe.
Smooth View-dependent Level-of-detail Control and Its Application To Terrain Rendering.
Available: IEEE Visualization proceedings 1998, pp. 35-42.

[102] H. Hoppe.
View-dependent Refinement Of Progressive Meshes.
Computer Graphics, SIGGRAPH proceedings 1997, pp. 189-198.

[103] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle.
Mesh Optimization.
Computer Graphics, SIGGRAPH proceedings 1993, pp. 19-26.

[104] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle.
Surface Reconstruction from Unorganized Points.
Available: Computer Graphics, SIGGRAPH proceedings 1992, pp. 71-81.

[105] J. Huang, R. Yagel, V. Filippov, Y. Kurzion.
An Accurate Method for Voxelizing Polygon Meshes.
ACM Symposium on Volume Visualization 1998, pp. 119-126.

[106] P. M. Hubbard.
Approximating Polyhedra With Spheres For Time-critical Collision Detection.
ACM Transactions on Graphics 15(3) 1996, pp. 179-210.

[107] P. M. Hubbard.
Interactive Collision Detection.
IEEE Symposium on Frontiers in Virtual Reality proceedings 1993, pp. 24-31.

[108] T. Hudson, D. Manocha, J. D. Cohen, M. C. Lin, K. Hoff, H. Zhang.
Accelerated Occlusion Culling Using Shadow Frusta.
Symposium on Computational Geometry 1997, pp. 1–10.

[109] C. L. Jackins, S. L. Tanimoto.
Oct-trees and Their Use in Representing Three Dimensional Objects.
Computer Graphics and Image Processing 14(3) 1980, pp. 249–270.

[110] P. Jimenez, F. Thomas, C. Torras.
3D Collision Detection: A Survey.
Computers and Graphics 25(2) 2001, pp. 269-285.

[111] D. Johnson, E. Cohen.
Spatialized Normal Cone Hierarchies.
Symposium on Interactive 3D Graphics proceedings 2001, pp. 129-134.

[112] I. Jolliffe.
Principle Component Analysis.
Springer-Verlag 2003, ISBN: 0387954422.

[113] A. Kalaiah, A. Varshney.
Differential Point Rendering.
Eurographics 12'th Workshop on Rendering proceedings 2001, pp. 139-150.

[114] A. Kalaiah, A. Varshney.
Statistical Point Geometry.
ACM SIGGRAPH Symposium on Geometry Processing 2003, pp. 107-115.

[115] S. B. Kang.
A Survey of Image-Based Rendering Techniques.
Cambridge Research Laboratory Technical Report CRL 97/4 1997.

[116] A. Kaufman.
An Algorithm For 3D Scan-conversion Of Polygons.
Eurographics proceedings 1987, pp. 197-208.

[117] A. Kaufman
Voxels As A Computational Representation of Geometry.
The Computational Representation of Geometry, SIGGRAPH course notes 1994.

[118] A. Kaufman, E. Shimony
3D Scan Conversion Algorithms for Voxel Based Graphics.
ACM Workshop on Interactive 3D Graphics proceedings 1986, pp. 45-76.

[119] S. Kilthau, T. Moller.
Splatting Optimizations.
Technical Report SFU-CMPT-04/01-TR2001-02,2001, Graphics, Usability and Visualization Lab, Simon Fraser University Ca.

[120] J. Klosowski, C. Silva.
Efficient Conservative Visibility Culling Using The Prioritized-Layered Projection Algorithm.
IEEE Transactions on Visualization and Computer Graphics 7(4) 2001, pp. 365-379.

[121] J. Klosowski, C. Silva.
The Prioritized Layered Projection Algorithm for Visible Set Estimation.
IEEE Transactions on Visualization and Computer Graphics 6(2) 2000, pp. 108-123.

[122] J. Koenderink, A. Van Doorn.
The Internal Representation of Solid Shape with Respect to Vision.
Biological Cybernetics 32(4), pp. 211-216.

[123] J. Krivanek.
Representing and Rendering Surfaces With Points.
Postgraduate Study Report DC-PSR-2003-03 2003, Czech Technical University, Prague.

[124] J. Krüger, J. Schneider, R. Westermann.
DUODECIM: A Structure for Point Scan Compression and Rendering.
IEEE Eurographics Symposium on Point-Based Graphics Proceedings 2005, pp. 125-126.

[125] S. Kumar, D. Manocha, B. Garrett, M. Lin.
Hierarchical Back-face Computation.
Eurographics 7'th Workshop on Rendering 1996, pp. 231-240.

[126] R. Lau, D. To, M. Green.
An Adaptive Multiresolution Modeling Technique Based on Viewing and Animation Parameters.
IEEE Virtual Reality Annual International Symposium proceedings 1997, pp. 20-27.

[127] D. Laur, P. Hanrahan.
Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering.
Computer Graphics, SIGGRAPH proceedings 1991, pp. 285-288.

[128] D. Levin.
Mesh-Independent Surface Interpolation.
Geometric Modeling for Scientific Visualization, Springer-Verlag 2003, ISBN: 3540401164, pp. 37-49.

[129] M. Levoy, P. Hanrahan.
Light Field Rendering.
Computer Graphics, SIGGRAPH 1996 proceedings, pp. 31-42.

[130] M. Levoy, T. Whitted.
The Use of Points as a Display Primitive.
Technical Report 85-022, 1985, Computer Science, University of North Carolina, Chapel Hill.

[131] M. C. Lin, S. Gottschalk.
Collision Detection Between Geometric Models: A Survey.
IMA Conference on Mathematics of Surfaces proceedings 1988, pp. 602-608.

[132] P. Lindstrom.
Model Simplification Using Image and Geometry- Based Metrics.
PhD Thesis 2000, Computer Science, Georgia Institute of Technology.

[133] P. Lindstrom, G. Turk.
Fast and Memory Efficient Polygonal Simplification.
IEEE Visualization proceedings 1998, pp. 279-286.

[134] P. Lindstrom, G. Turk.
Image-driven Simplification.
ACM Transactions on Graphics 19(3), pp. 204-241.

[135] A. Lippman.
Movie Maps: An Application of the Optical Videodisc to Computer Graphics.
Computer Graphics, SIGGRAPH proceedings 1980, pp. 32-43.

[136] D. Luebke, C. Erikson.
View-dependent Simplification of Arbitrary Polygonal Environments.
Computer Graphics, SIGGRAPH proceedings 1997, pp. 199–208.

[137] D. Luebke, C. Georges.
Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets.
Computer Graphics, SIGGRAPH proceedings 1995, pp. 105-106.

[138] D. Luebke, B. Hallen.
Perceptually Driven Simplification for Interactive Rendering.
Eurographics Workshop on Rendering Techniques proceedings 2001, pp. 223-234.

[139] D. Luebke, B. Hallen, D. Newfield, B. Watson.
Perceptually Driven Simplification Using Gaze-Directed Rendering.
Technical Report CS-2000-04, 2000, University of virginia.

[140] P. Maciel, P. Shirley.
Visual Navigation of Large Environments Using Textured Clusters.
ACM Symposium on Interactive 3D Graphics 1995, pp. 95-102.

[141] S. Mantler, A. Fuhrmann.
Point Based Rendering for Massive Data Sets: A Case Study.
IEEE Computer Graphics International 2004, pp. 182-187.

[142] W. Mark, L. McMillan, G. Bishop.
Post-Rendering 3D Warping.
ACM Symposium on Interactive 3D Graphics proceedings 1997, pp. 7-16.

[143] S. Martello, P. Toth.
Knapsack Problems : Algorithms and Computer Implementations.
Published by Wiley, 1999.

[144] A. E. W. Mason, E. H. Blake.
Automatic Hierarchical Level of Detail Optimization in Computer Animation.
Computer Graphics Forum, 16(3) 1997, pp. 191–200.

[145] T. Massie, K. Salisbury.
The PHANToM Haptic Interface: A Device for Probing Virtual Objects.
ASME Symp. on Haptic Interfaces and Virtual Environments and Teleoperator Systems 1994, pp. 295-301.

[146] L. McMillan, G. Bishop.
Plenoptic Modeling: An Image-Based Rendering System.
Computer Graphics, SIGGRAPH proceedings 1995, pp. 39-46.

[147] C. Mendoza, C. O'Sullivan.
Interruptible Collision Detection for Deformable Objects.
Computers and Graphics 30(3) 2006, pp. 432-438.

[148] G. Muller, D. Fellner.
An Adaptive Hierarchical Occlusion Culling Algorithm for Interactive Large Model Visualization.
Technical Report TUBS-CG-2001-05 2001, University of Technology, Braunschweig.

[149] B. Naylor.
Partitioning Tree Image Representation and Generation From 3D Geometric Models.
Graphics Interface proceedings 1992, pp. 201-212.

[150] M. M. Oliveira.
Image-Based Modeling and Rendering Techniques: A Survey.
RITA - Revista de Informática Teórica e Aplicada, 9(2) 2002, pp. 37-66.

[151] R. Pajarola.
Confetti : Object-Space Point Blending and Splatting.
IEEE Transactions on Graphics 10(5) 2004, pp. 598-608.

[152] R. Pajarola.
Efficient Level-of-Details for Point Based Rendering.
IASTED Computer Graphics and Imaging proceedings 2003.

[153] M. Pauly, M. Gross.
Spectral Processing of Point-Sampled Geometry.
Available: Computer Graphics, SIGGRAPH proceedings 2001, pp. 379-386.

[154] M. Pauly, M. Gross, L. P. Kobbelt.
Efficient Simplification of Point-sampled Surfaces.
IEEE Visualization proceedings 2002, pp. 163-170.

[155] M. Pauly, R. Keiser, M. Gross.
Multi-scale Feature Extraction on Point-sampled Surfaces.
Computer Graphics forum 22(3) 2003, pp. 281-290.

[156] M. Pauly, R. Keiser, L. P. Kobbelt, M. Gross.
Shape Modeling With Point-sampled Geometry.
ACM Transactions on Graphics 22(3), pp. 641-650.

[157] H. Peitgen, D. Saupe (Editors).
The Science of Fractal Images.
Available: Published by Springer-Verlag 1988, ISBN: 0-387-96608-0.

[158] H. Pfister, M. Zwicker, J. van Baar, M. Gross.
Surfels: Surface Elements as Rendering Primitives.
Computer Graphics, SIGGRAPH proceedings 2000, pp. 335–342.

[159] V. Popescu.
Forward Rasterization: A Reconstruction Algorithm for Image-Based Rendering.
PhD. Dissertation, Department of Computer Science, University of North Carolina 2001, TR01-019.

[160] J. Popovic, H. Hoppe.
Progressive Simplicial Complexes.
Computer Graphics, SIGGRAPH proceedings 1997, pp. 217-224.

[161] J. Purbrick, C. Greenhalgh.
Extending Locales: Awareness Management in Massive-3.
IEEE Virtual Reality Annual International Symposium proceedings 2000, pp. 287-287.

[162] S. Quinlan.
Efficient Distance Computation Between Non Convex Objects.
International Conference on Robotics and Automation proceedings 1994, pp. 3324-3329.

[163] J. Rasanen.
Surface Splatting: Theory, Extensions and Implementation.
Masters Thesis 2002, Department of Computer Science, Helsinki University of Technology.

[164] V. Rayward-Smith, H. I. Osman, C. R. Reeves, G. D. Smith.
Modern Heuristic Search Methods.
Published by Wiley 1996, ISBN 0-471-96280-5.

[165] M. Reddy.
Reducing Lags in Virtual Reality Systems Using Motion-sensitive Level Of Detail.
UK VR-SIG 2nd Conference proceedings 1994, pp. 25-31.

[166] M. Reddy.
A Survey of Level of Detail Support in Current Virtual Reality Solutions.
Virtual Reality Research, Development and Application (1)2 1995, pp. 95-98.

[167] W. Reeves.
Particle Systems - A Technique for Modeling a Class of Fuzzy Objects.
Computer Graphics, SIGGRAPH proceedings 1983, pp. 359-375.

[168] L. Ren, H. Pfister, M. Zwicker.
Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering.
Computer Graphics Forum, 21(3) 2002, pp. 461-470.

[169] P. Reuter, I. Tobor, C. Schlick, S. Dedieu.
Point-based Modelling and Rendering using Radial Basis Functions.
ACM Graphite Proceedings 2003, pp. 111-118.

[170] J. Rossignac, P. Borrel.
Multi-resolution 3D Approximations for Rendering Complex Scenes.
Modeling in Computer Graphics, published by Springer 1993, ISBN 0-387-56529-9, pp. 455-465.

[171] S. Rusinkiewicz, M. Levoy.
QSplat: A Multiresolution Point Rendering System for Large Meshes.
Computer Graphics, SIGGRAPH proceedings 2000, pp. 343–352.

[172] S. Rusinkiewicz, M. Levoy.
Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models.
ACM Symposium on Interactive 3D Graphics proceedings 2001, pp. 63-68.

[173] P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, J. Snyder.
Silhouette Clipping.
Computer Graphics, SIGGRAPH proceedings 2000, pp. 327-334.

[174] P. V. Sander, J. Snyder, S. J. Gortler, H. Hoppe.
Texture Mapping Progressive Meshes.
Computer Graphics, SIGGRAPH proceedings 2001, pp. 409-416.

[175] D. H. Sanders, R. K. Smidt.
Statistics: A First Course (sixth edition).
McGraw Hill 2000, ISBN 0-07-117753-1.

[176] M. Sattler, R. Sarlette, T. Mucken, R. Klein.
Exploitation of Human Shadow Perception for Fast Shadow Rendering.
ACM Symposium on Applied Perception in Graphics and Visualization proceedings 2005, pp. 131-134.

[177] G. Schaufler.
Dyamically Generated Imposters.
GI Workshop on Modeling, Virtual Worlds, Distributed Graphics proceedings 1995, pp. 129–136.

[178] G. Schaufler.
Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes.
Eurographics Workshop on Rendering proceedings 1997, pp. 151-162.

[179] G. Schaufler, J. Dorsey, X. Decoret, F. Sillion.
Conservative Volumetric Visibility With Occluder Fusion.
Computer Graphics, SIGGRAPH proceedings 2000, pp. 229-238.

[180] D. Schmalstieg.
Lodestar: An Octree-Based Level Of Detail Generator for VRML.
Second Symposium on the Virtual Reality Modeling Language 1997 proceedings, pp. 125-132.

[181] D. Schmalstieg, G. Schaufler.
Smooth Levels of Detail.
IEEE VRAIS proceedings 1997, pp. 12-19.

[182] R. D. Schraft, J. Neugebauer, T. Flaig, R. Dainghaus.
A Fuzzy Controlled Rendering System for Virtual Reality Systems Optimised by Genetic Algorithms.
Eurographics Workshop on Virtual Environments proceedings 1995, pp. 22–32.

[183] M. Segal, C. Korobkin, R. Van Widenfelt, J. Foran, P. Haeberli.
Fast Shadows and Lighting Effects Using Texture Mapping.
Computer Graphics, SIGGRAPH proceedings 1992, pp. 249-252.

[184] J. Shade, S. Gortler, Li-wei He, R. Szeliski.
Layered Depth Images.
Computer Graphics, SIGGRAPH proceedings 1998, pp. 231-242.

[185] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, J. Snyder.
Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments.
Computer Graphics, SIGGRAPH proceedings 1996, pp. 75–82.

[186] L. Shirman, S. Abi.
The Cone of Normals Technique for Fast Processing of Curved Patches.
Eurographics proceedings 1993, pp. 261-272.

[187] H. Shum, S. B. Kang.
A Review of Image-Based Rendering Techniques.
IEEE/SPIE Visual Communications and Image Processing Proceedings 2000, pp. 2-13.

[188] Silicon Graphics Inc.
OpenGL Performer Programmer's Guide.
Silicon Graphics document 007-1680-100 1997.

[189] C. Silva, J. Mitchell, P. Williams.
An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes.
ACM Symposium on Volume Visualization Proceedings 1998, pp. 87-94.

[190] M. Slater.
Constant Time Queries Uniformly Distributed Points on A Hemisphere.
Journal of Graphics Tools 7(1) 2002, pp. 33-44.

[191] M. Slater, Y. Chrysanthou.
View Volume Culling Using A Probabilistic Caching Scheme.
ACM VRST proceedings 1997, pp. 71-78.

[192] M. Slater, M. Usoh, Y. Chrysanthou.
The Influence of Dynamic Shadows on Presence in Immersive Virtual Environments.
Eurographics Workshop on Virtual Environments 1995, pp. 8-19.

[193] H. Sowizral, K. Rushforth, M. Deering.
The java 3D API specification.
Addison Wesley 2000, ISBN 0201710412.

[194] M. Stamminger, G. Drettakis.
Interactive Sampling and Rendering for Complex and Procedural Geometry.
Eurographics 12'th Workshop on Rendering proceedings 2001, pp. 151-162.

[195] E. Stollnitz, T. DeRose, D. Salesin.
Wavelets for Computer Graphics.
Published by Morgan Kaufmann 1996, ISBN 1558603751.

[196] O. Sudarsky, C. Gotsman .
Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality.
Computer Graphics Forum 15(3) 1996, pp. 249-258.

[197] I. Sutherland.
Sketchpad: A Man-Machine Graphical Communication System.
Spring Joint Computer Conference Proceedings 1963, pp. 329-346.

[198] R. Szeliski, D. Tonnesen.
Surface Modeling With Oriented Particle Systems.
Computer Graphics, 26(2) 1992, pp. 185–194.

[199] J. O. Talton, N. A. Carr, J. C. Hart.
Voronoi Rasterization of Sparse Point Sets.
IEEE Eurographics Symposium on Point Based Graphics 2005, pp. 33-38.

[200] S. Teller.
Visibility Computation in Densely Occluded Polyhedral Environments.
Ph.D. thesis, UC Berkeley, Computer Science Dpt. 1992 TR #92/708.

[201] S. Teller, C. Sequin.
Visibility Preprocessing for Interactive Walktroughs.
Computer Graphics, SIGGRAPH proceedings 1991, pp. 61-69.

[202] I. Tobor, C. Schlick, L. Grisoni.
Rendering by Surfels.
Graphicon proceedings 2000, pp. 193-204.

[203] VRML Consortium Incorporated.
The Virtual Reality Modeling Language.
International Standard ISO/IEC 14772-1:1997.

[204] I. Wald, H. Seidel.
Interactive Ray Tracing of Point-based Models.
Eurographics Symposium on Point-Based Graphics proceedings 2005, pp. 9-16.

[205] M. Wand, M. Fischer, I. Peter, F. Meyer auf der Heide, W. Straßer.
The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes.
Computer Graphics, SIGGRAPH proceedings 2001, pp. 361-370.

[206] M. Wand, W. Straßer.
Multi-Resolution Rendering of Complex Animated Scenes.
Computer Graphics Forum, 21(3) 2002, pp. 483-491.

[207] B. Watson.
Evaluation of the Effects of Frame Time Variation on VR Task Performance.
IEEE VRAIS 1996, pp. 38-52.

[208] B. Watson.
Managing Level of Detail Through Peripheral Degradation: Effects on Search Performance with a Head-Mounted Display.
ACM Transactions on Computer Human Interaction 4(4) 1997, pp. 323-346.

[209] J. Wernecke.
The Inventor Mentor.
Addison Wesley 1994, ISBN 0201624958.

[210] L. A. Westover.
Footprint evaluation for volume rendering.
Computer Graphics 24(4) 1990, pp. 367-376.

[211] L. A. Westover.
Interactive Volume Rendering.
Volume Visualization Workshop proceedings 1989, University of North Carolina, pp. 9-16.

[212] T. Whitted.
An Improved Illumination Model for Shaded Display.
Communication of the ACM 23(6) 1980, pp. 343-349.

[213] M. Wicke, M. Teschner, M. Gross.
CSG Tree Rendering For Point-sampled Objects.
Computer Graphics and Applications, Pacific Conference on Volume Rendering 2004, pp. 160-168.

[214] L. Williams.
Casting Curved Shadows On Curved Surfaces.
Computer Graphics, SIGGRAPH proceedings 1978, pp. 270-274.

[215] P. Williams.
Visibility Ordering Meshed Polyhedra.
ACM Transactions on Graphics 11(2) 1992, pp. 103-126.

[216] M. Wimmer, P. Wonka, F. Sillion.
Point-based Impostors for Real-time Visualization.
Eurographics 12'th Workshop on Rendering proceedings 2001, pp. 163-174.

[217] A. Woo, P. Poulin, A. Fournier.
A Survey of Shadow Algorithms.
IEEE Computer Graphics and Applications 10(6) 1990, pp. 13-32.

[218] C. Woolley, D. Luebke, B. Watson.
Interruptible Rendering.
ACM Symposium on Interactive 3D Graphics 2002, pp. 143-151.

[219] J. Wu, L. Kobbelt.
Optimized Sub-Sampling of Point Sets for Surface Splatting.
Computer Graphics Forum 23(3) 2004, pp. 643-652.

[220] J. Wu, Z. Zhang, L. Kobbelt.
Progressive Splatting.
IEEE Eurographics Symposium on Point Based Graphics 2005, pp. 25-32.

[221] J. C. Xia, A. Varshney.
Dynamic View-Dependent Simplification for Polygonal Models.
IEEE Visualization proceedings 1996, pp. 335-344.

[222] H. Xu, B. Chen.
Activepoints: Acquisition, Processing and Navigation of Large Outdoor Environments.
Technical Report TR.03.02 2003, University of Minnesota.

[223] H. Xu, M. Nguyen, X. Yuan, B. Chen.
Interactive Silhouette Rendering for Point-Based Models.
IEEE Eurographics Symposium on Point-Based Graphics 2004, pp. 2-4.

[224] S. Yoon, B. Salomon, D. Manocha.
Interactive View-Dependent Rendering with Conservative Occlusion Culling in Complex Environments.
IEEE Visualization Proceedings 2003, pp. 163-170.

[225] C. Zhang, T. Chen.
A Survey on Image Based Rendering.
Electrical and Computer Engineering, Carnegie Mellon University, Technical Report AMP 03-03 June 2003.

[226] H. Zhang, K. Hoff.
Fast Backface Culling Using Normal Masks.
ACM Symposium on Interactive 3D Graphics Proceedings 1997, pp. 103-106.

[227] E. Zhang, G. Turk.
Visibility Guided Simplification.
ACM Conference on Visualization Proceedings 2002, pp. 267-274.

[228] H. Zhang, D. Manocha, T. Hudson, K. Hoff.
Visibility Culling Using Hierarchical Occlusion Maps.
Computer Graphics, SIGGRAPH proceedings 1997, pp. 77–88.

[229] M. Zwicker.
Continuous Reconstruction, Rendering and Editing of Point-sampled Surfaces.
Ph.D. Dissertation 2003, Swiss Federal Institute of Technology, ETH, Zurich, Germany.

[230] M. Zwicker, M. Pauly, O. Knoll, M. Gross.
Pointshop 3D: An Interactive System for Point-Based Surface Editing.
ACM Transactions on Graphics, 21(3) 2002, pp. 322-329.

[231] M. Zwicker, H. Pfister, J. VanBaar, M. Gross.
EWA volume splatting.
IEEE Visualization proceedings 2001, pp. 29-36.

[232] M. Zwicker, H. Pfister, J. van Baar, M. Gross.
Surface splatting.
Computer Graphics, SIGGRAPH proceedings 2001, pp. 371-378.

[233] M. Zwicker, J. Rasanen, M. Botsch, C. Dachsbacher, M. Pauly.
Perspective Accurate Splatting.
ACM Graphics Interface proceedings 2004, pp. 247-254.